

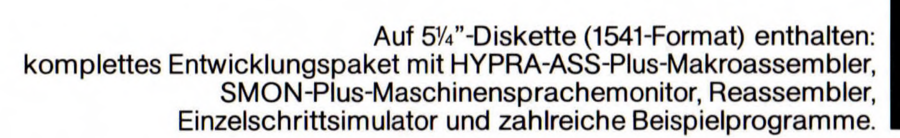


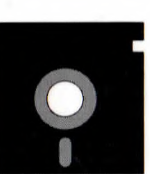
Frank Riemenschneider

C64/C128

Alles über Maschinensprache

Assemblerkurs ★ Interrupt-Programmierung
★ Variablenverwaltung und Fließkommarechnung
★ Grafikprogrammierung ★ Basic-Erweiterung


Auf 5¼"-Diskette (1541-Format) enthalten:
komplettes Entwicklungspaket mit HYPRA-ASS-Plus-Makroassembler,
SMON-Plus-Maschinensprachemonitor, Reassembler,
Einzelschrittsimulator und zahlreiche Beispielprogramme.



C64/C128 Alles über Maschinensprache



Frank Riemenschneider

C 64/C 128
Alles über Maschinensprache

Assemblerkurs ★ Interruptprogrammierung
★ Variablenverwaltung und Fließkommarechnung
★ Grafikprogrammierung ★ Basic-Erweiterung

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Riemenschneider, Frank:
C64, C128 : alles über Maschinensprache ; mit komplettem Assembler-Entwicklungssystem ; Assemblerkurs, Interrupt-Programmierung, Variablenverwaltung u. Fliesskommarechnung, Grafikprogrammierung, Basic-Erweiterung / Frank Riemenschneider. – Haar bei München : Markt-u.-Technik-Verl., 1988.
(Commodore-Sachbuch)
ISBN 3-89090-571-4

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
91 90 89 88

ISBN 3-89090-571-4

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Kapitel 3: Variablen in Maschinensprache 135

Kapitel 4: Programmierung der HiRes-Grafik 199

Kapitel 5: Programmierung von Basic-Erweiterungen 261

Anhang 277

Vorwort

Bei kaum einem anderen Computer ist die Diskrepanz zwischen der durch die Hardware gegebenen Leistungsfähigkeit und deren Ausnutzung durch das eingebaute Basic so groß wie beim C64.

So lassen sich viele Anwendungen erst durch die Programmierung in Maschinensprache realisieren, da der Basic-Interpreter wesentliche Nachteile aufweist: Zum einen kann man nur ca. 60 % des verfügbaren Speichers ausnutzen, wodurch z.B. so beliebte Programme wie Hi-Eddi oder Giga-CAD unmöglich gemacht würden.

Ein weiteres Kriterium stellt die Verarbeitungsgeschwindigkeit dar: Ein Maschinenprogramm kann bis zu 100mal schneller sein als ein entsprechendes in Basic. Wer schon einmal versucht hat, umfangreiche Datenmengen zu verwalten oder gar zu sortieren, kann hiervon ein Lied singen.

Als besonders nachteilig erweist sich jedoch der geringe Befehlsumfang des Basic-Interpreters. So lassen sich viele Programmiertechniken wie z.B. die Interruptprogrammierung überhaupt erst in Maschinensprache nutzen. Der Aufbau von Grafiken ist zwar theoretisch möglich, mit Hilfe der POKE-Befehle aber unzumutbar.

Leider meinen aber viele Computerbesitzer, daß die Maschinenprogrammierung zu schwer für sie sei und sie deshalb mit dem Basic vorlieb nehmen müßten, auch wenn sie dies auf Dauer nicht befriedigen könne. Dieses Vorurteil gilt aber nur dann, wenn man ein zu schwieriges Lehrbuch und kein gutes Maschinensprache-Entwicklungssystem besitzt.

In diesem Buch finden Sie eine umfangreiche und tabellarisch übersichtlich gestaltete Einführung in die Maschinensprache sowie auf der beiliegenden Diskette mit den Programmen Hypra-Ass Plus und SMON Plus eines der besten Assembler-Pakete für den C64. Damit läßt sich fast noch komfortabler programmieren als in Basic!

Der anwenderbezogene Teil dieses Buches beschäftigt sich mit vier Themengebieten, die man als »Spezialitäten« des C64 bezeichnen kann und die den Unterschied zwischen Basic- und Maschinenprogrammierung am deutlichsten aufzeigen. So werden im Kapitel »Interruptprogrammierung« Programmiertechniken vorgestellt, die man durch Basic überhaupt nicht nachvollziehen kann. Wie man u.a. mehrere hundert Datensätze statt in Stunden in Sekunden sortieren kann, erfahren Sie im Kapitel »Variablenverwaltung und Fließkommarechnung«. Natürlich darf auch die Grafikprogrammierung nicht fehlen. In diesem Kapitel wird aber auch

deutlich, daß Maschinensprache nicht gleich Maschinensprache ist. So sind die hier vorgestellten Routinen mehr als 10mal (!) schneller als die von vielen professionellen Grafikerweiterungen. Schließlich wird im Kapitel »Basic-Erweiterungen« auch darauf eingegangen, wie man dem mageren Basic durch Maschinenprogrammierung auf die Sprünge helfen kann. Abgerundet wird das Buch durch einen Anhang mit den wichtigsten Daten für den Maschinenprogrammierer.

Besonderer Wert wurde auf viele Programmbeispiele gelegt, die alle auf der beiliegenden Diskette abgelegt sind und von Ihnen auf einfachste Weise mit Hypra-Ass geändert werden können.

Ich hoffe jedenfalls, daß Ihnen dieses Buch den Schrecken der Maschinenprogrammierung nehmen kann und wünsche Ihnen viel Spaß bei der Lektüre.

Frank Riemenschneider

Ladehinweise zur beiliegenden Diskette

Bevor Sie beginnen, mit der Diskette zum Buch zu arbeiten, sollten Sie folgende Hinweise beachten.

Bitte laden Sie das Directory der Diskette mit

```
LOAD"$",8
```

und geben es mit

```
LIST
```

auf dem Bildschirm aus. Es erscheint folgendes Bild:

```
0  "90571     "mt          2a
0    "----------------"
0    "kapitel 1       "
0    "----------------"
38   "hypra assplus",8: prg
11   "makros"           prg
17   "smonpc000",8,1:   prg
17   "smonp3000",8,1:   prg
16   "smonic000",8,1:   prg
17   "smonfc000",8,1:   prg
0    "----------------"
0    "kapitel 2       "
0    "----------------"
29   "interr.-demo",8:  prg
8    "echtzeit-nmi"     prg
7    "timer-nmi"        prg
2    "system-irq"       prg
3    "rasterzeilen-irq" prg
5    "sprite-irq"       prg
5    "lightpen-irq"     prg
8    "abbruch-irq"      prg
0    "----------------"
0    "kapitel 3:      "
0    "----------------"
23   "wertetabelle"     prg
```

```
9    "bubblefliess"      prg
9    "bubbleinteger"     prg
12   "bubblestrings"     prg
0    "----------------"
0    "kapitel 4       "
0    "----------------"
8    "grafikroutinen"    prg
22   "plot"              prg
31   "rechteck"          prg
60   "kreis/ellipse"     prg
30   "text"              prg
58   "grafikdemo",8:     prg
0    "----------------"
0    "kapitel 5       "
0    "----------------"
30   "basicerweiterung"  prg
14   "grafik",8,1:       prg
0    "----------------"
11   "backup"            prg
184 blocks free.
```

Zunächst zu den Programmen, die keine Assemblerquelltexte darstellen. Man erkennt sie an dem Eintrag »,8:« bzw. »,8,1:« nach dem Filenamen. Um diese Programme zu laden, brauchen Sie also nur mit dem Cursor vor den Filenamen zu fahren und dann den Befehl

```
LOAD
```

mit anschließendem

```
RETURN
```

einzugeben. Wie die Programme gestartet werden, geht aus dem Text des jeweiligen Kapitels hervor.

Alle Programme ohne diesen Eintrag hinter dem Filenamen sind Quelltexte und werden vom Hypra-Ass Plus mit dem Befehl

```
/L"FILENAME"
```

geladen. Die Assemblierung wird mit

```
RUN
```

gestartet, worauf bis auf eine Ausnahme der Maschinencode direkt in den Speicher geschrieben wird.

Der Quelltext »KREIS/ELLIPSE« ist jedoch so lang, daß er sich bei der Assemblierung selbst überschreiben würde. Daher wird durch den Hypra-Ass-Plus-Befehl

```
.OB"KREIS/ELLIPSE.OB"
```

der Maschinencode nicht in den Speicher, sondern auf die Diskette geschrieben, die sich gerade im Laufwerk befindet. Den Maschinencode können Sie nun mit

```
LOAD"KREIS/ELLIPSE.OB",8,1
```

in den Speicher laden.

Backup

Auf keinen Fall sollten Sie mit der Original-Diskette arbeiten, sondern hierfür eine Sicherheitskopie benutzen. Hierzu befindet sich ein Kopierprogramm auf der Diskette zum Buch. Bitte laden Sie es mit

```
LOAD "BACKUP",8
```

und starten es mit

```
RUN
```

Auf dem Bildschirm erscheint die Einschaltmeldung. Sie werden nun gefragt, welche Tracks Sie kopieren möchten. Bitte machen Sie folgende Eingaben:

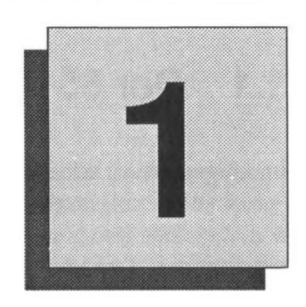
```
Starttrack: 4 (RETURN)
Endtrack: 34 (RETURN)
Alle Angaben richtig ? J (RETURN)
```

Das Kopierprogramm beginnt nun mit seiner Arbeit. Je nach Aufforderung des Programms müssen Sie die Original- und Ihre Zieldiskette ins Laufwerk einlegen und dies mit einem Druck der Return-Taste bestätigen.

Wenn die Meldung erscheint, daß der Kopiervorgang beendet ist, schalten Sie bitte Rechner und Floppy kurz aus und dann wieder ein. Dies ist sinnvoll, um beide Geräte neu zu initialisieren.

Die Original-Diskette legen Sie bitte an einen sicheren Platz, an dem sie nicht beschädigt werden kann. So können Sie jederzeit, falls Ihre Sicherheitskopie zerstört werden sollte, eine neue Kopie anfertigen.

Maschinensprache auf dem C64



Dies erste Kapitel soll Ihnen, liebe Leser, eine Einführung in die Maschinensprache geben und Ihnen dann unser Assembler-Entwicklungssystem vorstellen.

Zunächst möchte man fragen, gibt es eigentlich einen sachlichen Unterschied zwischen der Maschinensprache und dem Assembler ?

Nun, bei der reinen Maschinensprache handelt es sich um die »Muttersprache« des Prozessors unseres Computers. Sie ist extrem primitiv aufgebaut, denn sie besteht nur aus Nullen und Einsen. Im nächsten Abschnitt werden wir dies ausführlich erklären. Ein reines Maschinenprogramm stellt also nur eine Ansammlung von Einsen und Nullen dar. Um ein umfangreiches Programm zu entwickeln, müßte man sich mit mehr als 100.000 (!) solcher Zahlen herumquälen. Da selbst Albert Einstein wohl den Spaß am Computer verloren hätte, führte man die Assemblersprache ein. Diese kann man mit keiner Hochsprache wie Basic vergleichen. Der einzige Unterschied zur Maschinensprache besteht darin, daß bestimmte Folgen von Nullen und Einsen, die für unseren Prozessor eine besondere Bedeutung haben, durch sogenannte

Mnemonics (aus dem Griechischen: Die Kunst, das Gedächtnis durch Hilfen zu stärken)

dargestellt werden. Die Umwandlung von Assembler in Maschinensprache wird durch einen sogenannten Assembler (welch sinnvolle Bezeichnung...) durchgeführt. Hierbei handelt es sich um ein Programm, das »weiß«, welche Zahlenkombination welchem Mnemonic entspricht und dementsprechend diese Zahlen einsetzten kann. Die Trennung zwischen Maschinensprache und Assembler ist für uns als Anwender damit bedeutungslos geworden, wir werden natürlich nur in Assembler programmieren. Um den Zusammenhang zu verdeutlichen, möchte ich Ihnen ein kleines Beispiel vorführen, dessen Wirkung Sie an dieser Stelle natürlich noch nicht verstehen müssen. Es soll ausschließlich zeigen, wie angenehm sich der Assembler im Vergleich zur reinen Maschinensprache ausmacht:

Maschinensprache	Assembler	Wirkung auf Prozessor
101011010101100110001001	LDA $8959	Lädt sogenannten Akku mit Inhalt der Speicherzelle $8959.

1.1 Der 6510-Mikroprozessor

Im Gegensatz zu höheren Programmiersprachen, in denen man gute Programme auch dann entwickeln kann, wenn man den Aufbau des Prozessors kaum oder gar nicht kennt, ist es für den Maschinenprogrammierer unerläßlich, sich mit dem »Gehirn« des Computers genauer zu beschäftigen. Der Mikroprozessor 6510 des C64 besitzt eine Reihe von sogenannten Registern, in denen alle Rechenoperationen ablaufen. Um den Aufbau eines solchen Registers zu verstehen, muß man wissen, daß der Prozessor mit elektrischen Strömen arbeitet und daher überhaupt nur zwei Zustände unterscheiden kann: Entweder fließt Strom oder es fließt keiner. Als Programmierer hat man allerdings relativ wenig davon, zu wissen, ob Strom im Prozessor fließt. Deshalb rechnet man in dem sogenannten Dualsystem (Zahlensystem mit der Basis 2), das diese Zustände der Leitungen in die Mathematik übertragen kann, da auch hier genau zwei Zahlen existieren, nämlich die Null und die Eins. Zwischen Dualsystem und Prozessorströmen kann man demnach folgende Verbindung herstellen:

Dualzahl	Spannung	(V) Strom fließt	Elektrischer Pegel
0	0 – 0.8	nein	Low
1	2.4 – 5	ja	High

Ein Prozessorregister besitzt mit einer Ausnahme acht solcher »Schalter«, die man als Bits bezeichnet, da man mit einem Bit ja nur zwei verschiedene Werte charakterisieren könnte. Durch 8 Bit hingegen können 256 Zustände dargestellt werden. Dies ist nicht schwer zu verstehen, wenn man sich klarmacht, wie so ein Zahlensystem aufgebaut ist. Gewöhnlich rechnen wir im Dezimalsystem, dem Zahlensystem mit der Basis 10. Wenn wir z.B. die Zahl 156 betrachten, ist diese nur eine Abkürzung für den Rattenschwanz

1 * 10↑2 + 5 * 10↑1 + 6 * 10↑0

Jede weiter nach links gerückte Stelle ergibt einen Wert, der um den Faktor 10 größer ist als der vorhergehende, d.h. der Exponent wird um eins erhöht. Genauso wird im Zweiersystem verfahren, der Unterschied besteht nur in der Basis des Zahlensystems und darin, daß die Faktoren nur die Zahlen Eins und Null annehmen können, während im Dezimalsystem Zahlen von Null bis Neun vorkommen können. So können wir für die Binärzahl (wie die Zahlen des Dualsystems genannt werden) 11111111 schreiben:

1 * 2↑7+1 * 2↑6+1 * 2↑5+1 * 2↑4+1 * 2↑3+1 * 2↑2+1 * 2↑1+1 * 2↑0 = 255

Damit können Zahlen von Null bis 255 durch eine achtstellige Binärzahl angenommen werden, insgesamt also 256 verschiedene Werte, von den gesetzten Bits abhängig. Auf dem Papier findet man oft eine solche Zahl mit Angaben der Wertigkeiten der einzelnen Bits vor, um schneller den Wert errechnen zu können:

7	6	5	4	3	2	1	0	Bitposition
1	0	0	1	0	0	1	1	Bits

Für die Wertigkeit der einzelnen Positionen gilt dabei:

Bitposition	Exponent	Wert	Bitposition	Exponent	Wert
0	2↑0	1	4	2↑4	16
1	2↑1	2	5	2↑5	32
2	2↑2	4	6	2↑6	64
3	2↑3	8	7	2↑7	128

Leider ist es für den Programmierer sehr umständlich, sich durch solche Zahlenwüsten von Nullen und Einsen kämpfen zu müssen. Deshalb wurde das sogenannte Hexadezimalsystem (Zahlensystem mit der Basis 16) eingeführt, das genau dies Problem beseitigt. Die Idee besteht darin, jede 8-Bit-Binärzahl in zwei Teile, die sogenannten Nibbles, zu zerlegen. Jedes dieser Nibbles kann nun Werte von Null bis 15 annehmen, da die höchste Zweierpotenz jedes Nibbles ja nicht mehr 2↑7, sondern nur noch 2↑3 ist. Dadurch ergibt sich der maximale Nibble-Inhalt zu 2↑0 + 2↑1 + 2↑2 + 2↑3 = 1+2+4+8 = 15

Dezimal	Binär	Hexadezimal	Dezimal	Binär	Hexadezimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Jedem dieser Nibbles wird nun eine Hexadezimalzahl zugeordnet und diese beiden Zahlen aneinandergefügt. Da man nur die Zahlen von Null bis Neun zur Verfügung hat, muß man für die Zahlen 10–15 eine andere Lösung finden. Diese besteht darin, einfach hierfür die Buchstaben von A bis F zu benutzen. Die Tabelle auf der vorhergehenden Seite zeigt den Zusammenhang zwischen den einzelnen Zahlensystemen:

Um z.B. die Binärzahl 11110010 ins hexadezimale Format umzurechnen, geht man wie oben beschrieben vor: Nach der Aufteilung in die Nibbles 1111 und 0010 liest man die entsprechenden Hexadezimalzahlen F und 2 ab und setzt diese zusammen: F2. Um eine 8-Bit-Binärzahl in diesem Format darzustellen, sind also nur zwei Stellen erforderlich. Um die einzelnen Zahlensystemen zu unterscheiden, stellt man der Binärzahl ein %-Zeichen und der Hexadezimalzahl ein $-Zeichen vorweg, während die Dezimalzahl unverändert bleibt. Da die 8-Bit-Binärzahl für den Prozessor sehr wichtig ist, führte man den kurzen Begriff Byte für sie ein.

Ein weiteres Problem für uns besteht darin, auch andere Zeichen verarbeiten zu können, wie z.B. Buchstaben. Unser Prozessor weiß natürlich überhaupt nicht, was ein Buchstabe ist, er selbst kann ja nur Stromzustände unterscheiden. Deshalb führte man den sogenannten ASCII-Code ein, in dem jedem Zeichen eine bestimmte Zahl zwischen Null und 127 zugeordnet wird. ASCII ist dabei die Abkürzung für American Standard Code for Information Interchange und benötigt daher nur 7 Bit. So wird dem Zeichen »A« z.B. der Wert 65 zugewiesen, dem Zeichen »B« die 66, usw. Leider weicht der vom C64 verwendete »ASCII-Code« von dem Standard ab, was jedoch so lange bedeutungslos ist, bis man Daten mit anderen Rechnertypen austauschen möchte.

Unser Prozessor enthält fünf 8-Bit-Register sowie die schon erwähnte Ausnahme, ein 16-Bit-Register. Sehen wir uns nun einmal den Aufbau und die Funktion der einzelnen Register an:

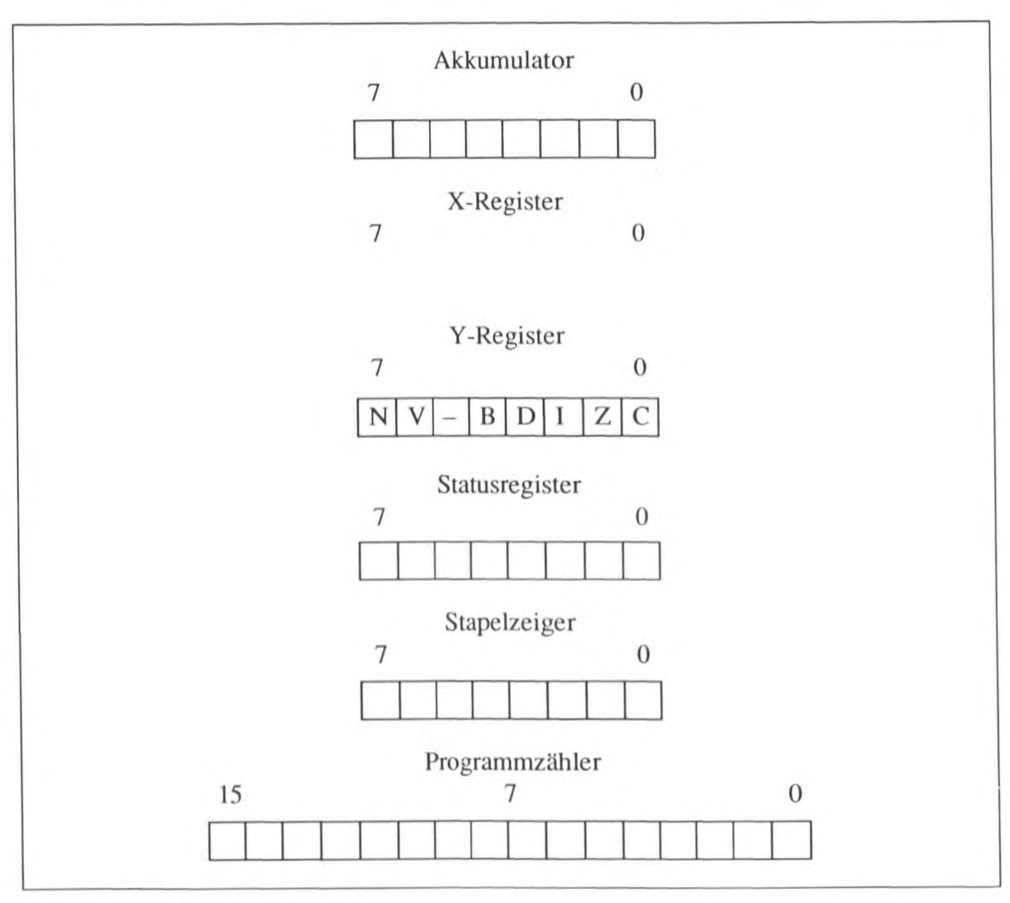


Bild 1.1: *Aufbau des Prozessors*

Der Akkumulator ist das wichtigste Register des Prozessors. In ihm laufen alle arithmetischen und logischen Operationen sowie fast alle Vergleiche ab. Die meisten Adressierungsarten der Befehle, zu denen wir noch kommen werden, lassen sich nur in Verbindung mit dem Akkumulator durchführen.

Das X-Register, auch oft als Indexregister bezeichnet, wird hauptsächlich dazu verwendet, als Index für den Akkumulator, z.B. bei der Abarbeitung von Tabellen, gute Dienste zu leisten. Weiterhin können auch Vergleichsoperationen durchgeführt werden. Als einziges Register des Prozessors kann man es dazu benutzen, den Stapelzeiger zu verändern, dazu jedoch später.

Das Y-Register dient ähnlichen Zwecken wie das X-Register. Auch hiermit lassen sich Vergleiche anstellen, den Stapelzeiger kann man jedoch nicht verändern.

Das Stackpoint-Register, auch Stapelzeiger genannt, enthält einen 8-Bit-Zeiger auf den sogenannten Stack. Hierbei handelt es sich um einen besonderen Adreßbereich von $0100–$01FF (dezimal: 256–511), der vom Prozessor als Speicher für wichtige Daten wie Rücksprungadressen von Unterprogrammen verwendet wird. Sicherlich wird Ihnen schon aufgefallen sein, daß sich der Bereich mit einem 8-Bit-Wert gar nicht ansprechen läßt, vielmehr werden 16 Bit benötigt. Da das höherwertige Adreßbyte mit $01 einen konstanten Wert aufweist, kann es vom Prozessor automatisch beigesteuert werden, der Stapelzeiger braucht sich nur um das niederwertige Adreßbyte kümmern. Nun zur Frage der Stackverwaltung. Der Zeiger des Stapelregisters zeigt immer auf die nächste freie Adresse des Stacks, und zwar abwärts gerechnet. Ist der ganze Stapel frei, enthält das Register daher den Wert $FF. Wenn nun ein Wert gespeichert werden soll, wird dieser an der Adresse $01FF abgelegt und, was ganz wichtig ist, der Stapelzeiger um eins auf den Wert $FE erniedrigt. Wäre dies nicht der Fall, würde die nächste Speicherung ja wieder an der Stelle $01FF erfolgen und damit der alte Wert überschrieben. Beim Lesen wird genau der umgekehrte Weg beschritten: Zunächst wird der Stapelzeiger um eins erhöht, um dann auf den letzten gespeicherten Wert zu zeigen. Dieser kann anschließend gelesen werden. Wenn wieder der Ausgangswert $FF erreicht wurde, ist der Stapel leer, d.h., er enthält keine brauchbaren Werte mehr.

Aus diesem Verfahren ergibt sich, daß man immer nur den zuletzt gespeicherten Wert auslesen kann. Will man daher »per Hand« stapeln, was durchaus möglich ist, muß man genau überlegen, welche Werte man zuerst ablegt, da man nicht so ohne weiteres an sie herankommt. Es gibt jedoch die Möglichkeit, den Stackpointer mit Hilfe des X-Registers zu verändern. Dieses Verfahren ist jedoch hochgradig gefährlich und nur dem fortgeschrittenen Programmierer zu empfehlen. Bild 1.2 erläutert das Stapelverfahren.

Das Statusregister enthält 7 sogenannte Flags, die vom Prozessor gelöscht oder gesetzt werden können. Sie geben Auskunft über das Ergebnis des letzten ausgeführten Befehls und sind einfach abfragbar. Viele Sprungbefehle basieren z.B. auf dem Zustand eines bestimmten Flags. Über die vielfältigen Möglichkeiten werden wir noch genau sprechen, deshalb soll hier zunächst nur kurz der Verwendungszweck angerissen werden:

Schritt	Operation	Stackpointer	Adresse	Stackinhalt
1	keine	$FF	$01FF	undefiniert
2	Schieben	$FE	$01FF $01FE	Wert 1 undefiniert
3	Schieben	$FD	$01FF $01FE	Wert 1 Wert 2
4	Holen	$FE	$01FD $01FF	undefiniert Wert 1
usw.			$01FE	undefiniert

Bild 1.2: *Nachführen des Stackpointers*

C – Carry: Das Carry-Flag dient dazu, anzuzeigen, ob bei einer Rechenoperation ein Über- oder Unterlauf aufgetreten ist, d.h. ob das Ergebnis in einem Bereich liegt, der sich nicht mehr durch ein Byte darstellen läßt.

Z – Zero: Das Zero-Flag (=Null-Flag) wird vom Prozessor immer dann gesetzt, wenn das Ergebnis einer Operation Null ist.

I – Interrupt: Dieses Flag bestimmt, ob ein sogenannter maskierbarer Interrupt erlaubt ist, den wir im Kapitel »Interruptprogrammierung« kennenlernen werden.

D – Dezimal: Neben dem normalen Binärmodus kann der Prozessor auch im Dezimalmodus rechnen, der ebenfalls im Kapitel über die Interrupts angesprochen wird.

B – Break: Das Break-Flag zeigt an, ob der Prozessor auf den Break(=Unterbrechung)-Befehl gestoßen ist. Er ist ebenfalls Thema der Interruptprogrammierung.

V – Overflow: Durch dieses Flag werden Über- bzw. Unterläufe beim Rechnen mit vorzeichenbehafteten Zahlen angezeigt. Es soll zunächst nicht interessieren.

N – Negative: Schließlich zeigt das Negativ-Flag an, ob das Ergebnis einer Operation größer als 127 ist, ob also das 7. Bit gesetzt ist. Eine Bedeutung erlangt es dann, wenn man Zahlen, die größer als 127 sind, als negativ interpretiert, was eine durchaus sinnvolle Vereinbarung sein kann.

Als letztes Register kommen wir nun auf den Programmzähler zu sprechen. Da unser Prozessor allein natürlich hilflos ist, müssen wir ihm die Informationen, d.h. die Befehle und die zu verarbeitenden Daten, liefern. Dafür stellt unser C64 64 Kbyte (1 Kbyte = 1024 Byte) an Speicher zur Verfügung. Jede Speicherzelle ist ebenfalls 8 Bit »groß«, so daß in ihr Zahlen von Null bis 255 gespeichert werden können. Damit ist ein einfacher Datenaustausch zwischen Speicher und Prozessorregistern möglich. Um diese 65536 Zellen auch ansprechen zu

können, genügt kein 8-Bit-Wert mehr. Vielmehr sind 16 Bit erforderlich (2^16 = 65536). Dies ist der Grund dafür, daß unser Programmzähler doppelt so groß ist wie die übrigen Register. In ihm befindet sich immer die Adresse der Speicherzelle, aus der sich der Prozessor den nächsten Befehlscode holt. Diese Adresse ist im 2-Byte-Format gespeichert, in der Form Lowbyte und Highbyte, die jeweils 8 Bit belegen. Mit dem Lowbyte bezeichnet man dabei den niederwertigen und mit dem Highbyte den höherwertigen Adreßteil der Speicherzelle. Wenn also als nächstes z.B. die Adresse $467D angesprochen werden sollte, enthielte unser Programmzähler die Bits

15 8	7 0	Position
01111101	01000110	Bits
Low=$7D	High=$46	

1.2 Die Adressierungsarten des 6510

Wie im letzten Abschnitt festgestellt wurde, kann jede Speicherzelle einen 8-Bit-Wert aufnehmen. Damit stehen für unseren Prozessor theoretisch 256 verschiedene Befehle zur Verfügung. Tatsächlich existieren jedoch nur 56 reguläre Befehle und einige wenige sogenannte »illegale Opcodes«, die wir später besprechen werden. Hätte man nur diese 56 Befehle zur Verfügung, könnte man kaum ein Maschinenprogramm entwerfen ohne wahrscheinlich hinterher in eine Irrenanstalt eingewiesen zu werden. Erst durch die verschiedenen Adressierungsarten der einzelnen Befehle ergeben sich insgesamt 151 verschiedene Kombinationen, mit denen man recht gut leben kann, auch wenn man zugeben muß, daß sich z.B. ein Z80-Prozessor wesentlich komfortabler programmieren läßt.

Zunächst möchte ich Ihnen nun die verschiedenen Adressierungen an Hand des Maschinenbefehls LDA nahebringen. Dieser dient dazu, den Akku mit einem Wert zu laden. Bis auf vier Arten kann man alle Adressierungsmöglichkeiten mit dem LDA-Befehl erschlagen. Zunächst jedoch zu den übrigen vier Adressierungen. Wichtig hierbei ist nicht, daß Sie die genaue Wirkungsweise der Befehle verstehen, da diese später ausführlich erläutert wird. Es kommt vielmehr darauf an, prinzipiell die einzelnen Adressierungen zu unterscheiden. Beim LDA-Befehl wurden die Beispiele durch die entsprechenden Basic-Analogons verdeutlicht.

1.2.1 Die implizite Adressierung

Beispiel: `INX`

Hierbei handelt es sich um Ein-Byte-Befehle, die die Prozessorregister verändern. Dabei bedeutet implizit, daß der Operand im Befehl mit enthalten ist. Durch den Befehl INX z.B. wird das X-Register um eins erhöht, der Wert Eins wird durch den eigentlichen Befehl vorgegeben.

1.2.2 Die Akkumulator-Adressierung

Beispiel: `LSR A`

Diese Befehle wirken direkt auf den Akku ein und stellen auch eine Art implizite Adressierung dar, weil die Wirkung durch den Befehl vorgegeben ist. So läßt der LSR-Befehl alle Bits des Akku um eine Stelle nach rechts verschieben.

1.2.3 Die relative Adressierung

Beispiel: `BCS #02`

Die relative Adressierung stellt diverse Sprungbefehle zur Verfügung, deren Zieladresse nicht fest ist, sondern vom Ausgangspunkt durch einen Offset berechnet wird. Hierbei handelt es sich um einen 2-Byte-Befehl, der aus dem eigentlichen Befehlscode und dem Offset besteht. Die Wirkung der 1-, 2- und 3-Byte-Befehle auf den Programmzähler sowie die Wirkung der einzelnen Offsets werden später erläutert werden.

1.2.4 Die indirekt-absolute Adressierung

Beispiel: `JMP ($A000)`

Hierbei handelt es sich um Sprungbefehle, die die Zieladresse aus den dem Befehlscode folgenden zwei Speicherzellen auslesen, in der Reihenfolge Lowbyte und Highbyte. Damit handelt es sich um einen 3-Byte-Befehl. Wenn z.B. die Speicherzelle $A000 den Wert $D5, und die Zelle $A001 den Wert $FF enthält, wird demnach die Adresse $FFD5 angesprungen.

1.2.5 Die unmittelbare Adressierung

Beispiel: `LDA #$10`
Basic-Analogon: `A = $10`

Bei der unmittelbaren Adressierung wird der Akku mit dem Wert geladen, den die auf den Befehlscode folgende Speicherzelle enthält. Die »$10« ist also selbst Bestandteil des Programms. Wenn der Prozessor nun auf einen Befehlscode eines solchen 2-Byte-Befehls trifft, interpretiert er automatisch den Inhalt der folgenden Adresse als Wert, den er in den Akku laden soll. Nach der Ausführung des Befehls wird der Programmzähler deshalb auch um zwei erhöht, da ja sonst die »10« als neuer Maschinenbefehl interpretiert würde, wenn er nur um eins erhöht würde. Das »#«-Zeichen zeigt an, daß der Wert »$10« und nicht der Inhalt der Speicherzelle $10 geladen werden soll.

1.2.6 Die absolute Adressierung

Beispiel: `LDA $FDE8`
Basic-Analogon: `A = PEEK($FDE8)`

Hierbei wird der Akku nicht mit einem konstanten Wert, sondern mit dem Inhalt einer bestimmten Speicherzelle geladen. Das Maschinenprogramm selbst muß diese Adresse nach dem Befehlscode in dem Format Lowbyte/Highbyte enthalten. Da auf diese Weise insgesamt drei Byte benötigt werden, wird der Programmzähler nach der Ausführung um drei erhöht, es handelt sich also um einen 3-Byte-Befehl.

1.2.7 Die Zeropage-Adressierung

Beispiel: `LDA $45`
Basic-Analogon: `A = PEEK($45)`

Eine besondere Rolle spielt der Adreßraum von $0000 – $00FF. Um ihn anzusprechen, wird nämlich nur ein Byte benötigt, da der höherwertige Adreßteil immer Null ist. Die Zeropage-Adressierung würdigt dies, indem ein spezieller Ladebefehl für diesen Bereich existiert. Der Vorteil besteht darin, daß er nur zwei Byte benötigt, da nach dem Befehlscode nur noch ein Adreßbyte folgt. Die Vorteile gegenüber der absoluten Adressierung liegen in der Platzersparnis und vor allen Dingen in einer schnelleren Verarbeitung. Auf diesen Punkt werden wir ebenfalls noch eingehen.

1.2.8 Die absolut-X-indizierte Adressierung

Beispiel: `LDA $879F,X`
Basic-Analogon: `A = PEEK($879F+X)`

Hierbei wird der Akku nicht mit dem Wert aus der Zelle $879F, sondern aus der Zelie geladen, deren Adresse sich aus der Summe von $879F und dem X-Register ergibt. Würde dies z.B. den Wert $12 enthalten, würde der Akku mit dem Inhalt der Speicherzelle $879F+12 = $87B1 geladen. Als sehr bedauerlich erweist sich in der Praxis, daß es zwar möglich ist, auch das Y-Register mit dem Befehl LDY $879F,X indiziert zu laden, nicht jedoch abzuspeichern. Dies bleibt dem Akku vorbehalten.

1.2.9 Die Zeropage-X-indizierte Adressierung

Beispiel: `LDA $F6,X`
Basic-Analogon: `A = PEEK($F6+X)`

Um beim Zugriff auf die Zeropage Platz und Zeit zu sparen, gibt es auch bei dieser Adressierungsart einen speziellen Befehl. Leider hat die Sache aber einen Haken: Da dieser Befehl als

2-Byte-Befehl ausschließlich auf die Zeropage zugreifen kann, wird ein eventueller Übertrag nicht berücksichtigt. Enthielte das X-Register z.B. den Wert $10, müßte normalerweise der Inhalt der Speicherzelle $F6+$10 = $0106 in den Akku geladen werden. Da sich diese Adresse jedoch außerhalb des Zugriffsgebietes bewegt, wird der Inhalt der Zelle $06 geladen, wodurch natürlich im allgemeinen ein völlig falscher Wert benutzt wird. An dieser Heimtücke ist schon so mancher Maschinenprogrammierer zugrunde gegangen. Im Zweifelsfall sollte man daher immer den 3-Byte-Befehl benutzen, hier also LDA $00F6,X.

1.2.10 Die absolut-Y-indizierte Adressierung

Beispiel: `LDA $1234,Y`
Basic-Analogon: `A = PEEK($1234+Y)`

Hierzu gilt das für die X-Indizierung Gesagte. Auch bei dieser Variante gibt es einen Ladebefehl für das X-Register, eine Y-indizierte Speicherung existiert dafür leider nicht.

1.2.11 Die Zeropage-Y-indizierte Adressierung

Beispiel: `LDX $F4,Y`
Basic-Analogon: `X = PEEK($F4+Y)`

Es gilt das unter dem Punkt »Zeropage X-indizierte Adressierung« Gesagte, inklusive der Fälle der Bereichsüberschreitung. Um diese in unserem Beispiel zu vermeiden, dürfte das Y-Register maximal den Wert $0B aufweisen, da $F4+$0B = $FF die höchste Adresse der Zeropage ist.

1.2.12 Die X-indiziert-indirekte Adressierung

Beispiel: `LDA ($85,X)`
Basic-Analogon: `A = PEEK((PEEK(85+X)+256*PEEK(86+X))`

Diese komplizierte Adressierung wird in der Praxis kaum gebraucht. Zunächst wird der Inhalt des X-Registers zu dem Operanden, hier $85, hinzuaddiert. Daraus ergibt sich ein Zeiger auf die Adresse, die nun selbst als Zeiger für die Adresse fungiert, aus der der Akkuinhalt ausgelesen wird. Falls z.B. das X-Register den Wert 3 aufwiese, würde der erste Zeiger auf die Adressen $88 (Lowbyte) und $89 (Highbyte) weisen. Die Inhalte dieser beiden Speicherzellen würden nun selbst die Speicherzelle bestimmen, deren Inhalt in den Akku geladen wird. Wenn z.B. die Zelle $88 den Inhalt $D2 und die Zelle $89 den Inhalt $FF aufwiesen, würde der Akku mit dem Inhalt der Zelle $FFD2 geladen. Es ist zu beachten, daß der Zeiger der Zellen $88/$89 auf den gesamten Adreßraum zeigen kann, der Zeiger selbst darf sich jedoch nur in der Zeropage befinden, so daß es sich auch nur um einen 2-Byte-Befehl handelt. Auch hier ist die oben geschilderte Bereichsüberschreitung zu vermeiden: Wenn die Summe von Operand und X-Register einen Wert größer als $FF ergibt, wird das höhere Adreßbyte igno-

riert und damit ein falscher Zeiger produziert. Da es diese Adressierung jedoch nicht im 3-Byte-Format gibt, muß man unbedingt aufpassen, daß das X-Register nicht einen zu hohen Wert annimmt.

1.2.13 Die indirekt-Y-indizierte Adressierung

Beispiel: `LDA ($FA),Y`
Basic-Analogon: `A = PEEK((Peek($FA)+256*PEEK($FB)+Y)`

Im Gegensatz zur letzten Adressierung wird diese in der Praxis sehr häufig benutzt. Der Grund hierfür ist, daß der Zeiger auf die Adresse, aus welcher der Akku geladen wird, in zwei feststehenden Speicherzellen der Zeropage zu finden ist, hier in den Zellen $FA (Lowbyte) und $FB (Highbyte). Der Zeiger verbleibt unabhängig vom Inhalt des Y-Registers immer in diesen beiden Speicherzellen. Erst wenn die Zieladresse ausgelesen wurde, wird der Inhalt des Y-Registers hinzuaddiert, um dadurch die endgültige Adresse zu erhalten. Falls also z.B. die Zellen $FA und $FB die Inhalte $D8 und $90 aufweisen sowie das Y-Register den Wert $A2 beinhaltet, wird der Akku mit dem Inhalt der Zelle $90D8 + $A2 = $917A geladen.

1.3 Übersicht und Funktion aller 6510-Befehle

In diesem Abschnitt werde ich Ihnen nun alle legalen und illegalen Befehle des 6510 vorstellen. Neben der genauen Erklärung der Wirkungsweise wird jeder Befehl in einer Tabelle der folgenden Form aufgelistet, aus der alle wichtigen Informationen sofort abzulesen sind:

Befehl:					Funktion:								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX													
Bytes													
Takte													
Flags N V B D I Z C					Beschreibung								

- Befehl: mnemonische Form des Befehls z.B. LDA
- Funktion: Der Pfeil zeigt auf das Ziel, z.B. M (M)+1

  Dabei bedeuten:
  A: Akkumulator
  X: X-Register
  Y: Y-Register

M: Speicher (Memory)
M6: Bit 6 des Inhalts der Speicherzelle
M7: Bit 7 des Inhalts der Speicherzelle
S: Stackpointerregister
PC: Programmzähler Low- und Highbyte
ADR: Adresse
P: Prozessorstatusregister
C: Carry-Flag
Z: Zero-Flag
I: Interrupt-Flag
D: Dezimal-Flag
B: Break-Flag
N: Negativ-Flag

Die Klammer weist auf den Inhalt der Quelle bzw. des Ziels hin

- Adress.: Die Adressierungsarten, die zur Verfügung stehen, wobei sie abgekürzt wiedergegeben sind. Die Reihenfolge entspricht der des vorhergehenden Abschnitts.
- HEX: Der Befehlscode in hexadezimaler Form. Ist eine Adressierungsart für einen bestimmten Befehl vorhanden, so ist die Spalte unter dieser mit zwei Byte ausgefüllt.
- Bytes: Anzahl der benötigten Bytes
- Takte: Anzahl der Taktzyklen, die zur Abarbeitung des Befehls benötigt werden. Ist die Zahl mit einem # markiert, so sind ein zusätzlicher Zyklus bei einer Verzweigung und zwei weitere bei einer Page-Überschreitung nötig.
- Flags: Befindet sich unter der Abkürzung eines Flags ein Eintrag, so wird dies durch den Befehl beeinflußt. Dies bedeutet:
  X: Flag wird vom Ergebnis der Operation beeinflußt.
  0: Flag wird gelöscht.
  1: Flag wird gesetzt
  6: Bit 6 des getesteten Bytes wird hineinkopiert.
  7: Bit 7 des getesteten Bytes wird hineinkopiert.
  *: Bit wird gesetzt, bevor das Statusregister auf den Stapel geschoben wird.

1.3.1 Die Ladebefehle

Den Befehl LDA zum Laden des Akkus kennen wir schon. Neben dem Akku ist es auch möglich, die Indexregister mit einem Wert zu laden. Die entsprechenden Befehle hierfür lauten

- LDX für das X-Register und
- LDY für das Y-Register.

Natürlich ist es auch möglich, die Inhalte dieser drei Register abzuspeichern. Dafür existieren die drei Befehle

- STA für den Akkumulator,
- STX für das X-Register und
- STY für das Y-Register.

Diese sechs Befehle sind die am meisten benutzten überhaupt.

Befehl: **LDA** Funktion: A ← Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					A9	AD	A5	BD	B9	B5		A1	B1
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X					X		Lade den Akkumulator mit neuen Daten.

Befehl: **LDX** Funktion: X ← Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					A2	AE	A6		BE		B6		
Bytes					2	3	2		3		2		
Takte					2	4	3		*4		4		

Flags	N	V	B	D	I	Z	C	
	X					X		Lade das X-Register mit neuen Daten.

Befehl: **LDY** Funktion: Y ← Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					A0	AC	A4	BC		B4			
Bytes					2	3	2	3		2			
Takte					2	4	3	*4		4			

Flags	N	V	B	D	I	Z	C	
	X					X		Lade das Y-Register mit neuen Daten. Der alte Inhalt wird gelöscht.

Bild 1.3: *Die LOAD-Befehle*

1.3.2 Die Speicherbefehle

Die folgenden 3 Befehle sind das Gegenstück zu den Ladebefehlen. Mit ihnen kann man die Registerinhalte des Akkumulators und der beiden Indexregister im Speicher ablegen. Im einzelnen sind dies

- STA zum Abspeichern des Akkus,
- STX zum Abspeichern des X-Registers und
- STY zum Abspeichern des Y-Registers.

Als Adressierungsarten stehen die gleichen wie bei den Ladebefehlen zur Verfügung, wobei eine unmittelbare Adressierung nicht möglich ist. Die Register selbst ändern sich beim Speichern nicht, weshalb auch keine Flags beeinflußt werden.

Befehl: **STA**	Funktion: M ← (A)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						8D	85	9D	99	95		81	91
Bytes						3	2	3	3	2		2	2
Takte						4	3	5	5	4		6	6

Flags N V B D I Z C	Speichere den Akkuinhalt, die Speicherzelle wird überschrieben.

Befehl: **STX**	Funktion: M ← (X)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						8E	86				96		
Bytes						3	2				2		
Takte						4	3				4		

Flags N V B D I Z C	Speichere den Inhalt des X-Registers, die Zelle wird überschrieben.

Bild 1.4: *Die STORE-Befehle STA und STX*

Befehl: **STY**						Funktion: M ← (Y)							
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						8C	84			94			
Bytes						3	2			2			
Takte						4	3			4			

Flags	N	V	B	D	I	Z	C	
								Speichere den Inhalt des Y-Registers, die Zelle wird überschrieben.

Bild 1.5: *Der STORE-Befehl STY*

1.3.3 Die Transferbefehle innerhalb des Prozessors

Der 6510-Mikroprozessor bietet 4 Befehle, um den Inhalt eines Indexregisters in den Akkumulator zu übertragen oder umgekehrt. Dabei bleibt der Wert des Ursprungsregisters erhalten. Die Bedeutung dieser Befehle ist deshalb so groß, weil viele Befehle nur mit dem Akku arbeiten, nicht aber mit den Indexregistern. Wenn man z.B. den Inhalt des X-Registers zu dem einer Speicherzelle hinzuaddieren möchte, müßte man das X-Register zwischenspeichern und dann in den Akku einladen, bevor man die Addition ausführen könnte. Eine weitere, häufig benutzte Verwendung liegt darin, eines der Indexregister als Zwischenspeicher für den Akku zu verwenden. Dieser Fall tritt dann ein, wenn ein im Akku errechnetes Ergebnis noch gebraucht wird, auf der anderen Seite jedoch sofort mit ihm weitergerechnet werden soll. Leider ist es nicht möglich, Daten der Indexregister untereinander auszutauschen. Alle Transferbefehle sind 1-Byte-Befehle, die jeweils nur zwei Taktzyklen benötigen. Wie bei den Ladebefehlen werden auch hier das Zero- und Negative-Flag beeinflußt. Hier nun die Befehle im einzelnen:

- TAX kopiert Akkuinhalt in X-Register
- TXA kopiert Inhalt des X-Registers in Akku
- TAY kopiert Akkuinhalt in Y-Register
- TYA kopiert Inhalt des Y-Registers in Akku

Befehl: **TAX**						Funktion: X ← (A)							
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	AA												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Kopiere den Akkuinhalt ins X-Register. Der Akku bleibt unverändert.

Bild 1.6: *Die Registerbefehle TAX, TXA und TAY (Fortsetzung nächste Seite)*

Befehl: **TXA**	Funktion: A ← (X)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	8A												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Kopiere X-Registerinhalt in den Akku. X-Register bleibt unverändert.

Befehl: **TAY**	Funktion: Y ← (A)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	A8												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Kopiere den Akkuinhalt ins Y-Register. Der Akku bleibt unverändert.

Bild 1.6: *Die Registerbefehle TAX, TXA und TAY (Ende)*

Befehl: **TYA**	Funktion: A ← (Y)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	98												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Kopiere Y-Registerinhalt in den Akku. Y-Register bleibt unverändert.

Bild 1.7: *Der Registerbefehl TYA*

Neben diesen vier Befehlen gibt es noch zwei weitere, die jedoch eine grundsätzlich andere Funktion aufweisen. Mit ihnen kann man die Registerinhalte von X-Register und, ja Sie lesen richtig, Stackpoint-Register austauschen. Diese lauten:

- TXS überträgt das X-Register in das Stackpoint-Register
- TSX überträgt das Stackpoint-Register in das X-Register

Dies ist der einzige Fall, in dem das X-Register selbst dem Universalgenie Akkumulator überlegen ist. Wozu aber braucht man diese Befehle? Nun, wichtig ist ja erst einmal, beim Einschalten des Computers das Stackpoint-Register mit dem Startwert $FF zu initialisieren. Dies wird in der Reset-Routine des C64 durchgeführt. Weiterhin bieten die Befehle die Möglichkeit, bei sehr tief verschachtelten Programmen schlagartig in das aufrufende Programm

zurückzuspringen. Man muß dazu wissen, daß bei jedem Aufruf eines Unterprogramms die Rücksprungadresse vom Prozessor automatisch auf den Stapel geschoben wird, bei der Rückkehr wird diese wieder gelesen. Bei drei verschachtelten Unterprogrammen befinden sich also insgesamt 6 Byte auf dem Stapel (jeweils Low- und Highbyte der Rücksprungadresse). Wenn man nun den Stackpointer um 4 erhöht, wird vom dritten Unterprogramm nicht mehr in das zweite Unterprogramm, sondern gleich in das Hauptprogramm zurückgesprungen.

Befehl: **TXS** Funktion: S ← (X)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	9A												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
								Kopiere X-Registerinhalt in den Stapelzeiger. Das X-Register bleibt gleich.

Befehl: **TSX** Funktion: X ← (S)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	BA												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Kopiere den Stapelzeiger ins X-Register. Dieser bleibt unverändert.

Bild 1.8: *Die X-Register-Stapelbefehle TSX und TXS*

1.3.4 Die arithmetischen Befehle

Der Sinn eines Computers besteht darin, Rechenoperationen in extrem kurzer Zeit ausführen zu können. Deshalb wäre er ohne die Rechenbefehle nur ein nutzloser Haufen von Elektronik. Wer aber nun glaubt, daß der 6510-Prozessor für alle mathematischen Operationen wie Quadratwurzelziehen etc. einen Befehl zur Verfügung stellt, irrt gewaltig. Er beherrscht nämlich nur die Addition und die Subtraktion, und dort immer nur eine Rechnung von 1-Byte-Werten! Alle weiteren mathematischen Funktionen müssen durch teilweise komplizierte Maschinenprogramme erstellt werden.

Der Additionsbefehl heißt

ADC

und addiert zu dem Inhalt des Akkumulators je nach Adressierungsart den Inhalt einer Speicherstelle oder bei der unmittelbaren Adressierung einen vorgegebenen Wert. Weiterhin wird der Inhalt des Carry-Flags automatisch hinzuaddiert. Den Sinn dieser Maßnahme werden wir gleich erläutern. Wie gesagt, kann man nur zwei Ein-Byte-Werte verknüpfen. Dies geschieht so lange problemlos, bis ein Überlauf entsteht. Hierzu zwei Beispiele: Zunächst wollen wir annehmen, der Akku enthielte den Wert $56. Durch den Befehl

```
ADC #$34
```

wird folgende Rechnung ausgeführt:

```
  $56 = % 01010110
+ $34 = % 00110100
------------------
= $8A = % 10001010
```

Wie bei der dezimalen Addition werden auch hier die einzelnen Stellen verknüpft. Dabei können 3 verschiedene Fälle auftreten:

0 + 0 = 0
0 + 1 = 1
1 + 1 = 0 plus Übertrag

Tritt ein Übertrag auf, so wird dieser bei der folgenden Stelle mit berücksichtigt. Dadurch ist das obige Ergebnis entstanden, da bei unserer Rechnung insgesamt drei Überträge auftraten. Man sieht, daß das Ergebnis sich ohne Probleme durch 8 Bit darstellen läßt. Anders sieht jedoch der Fall aus, wenn der Akku den Wert $E2 enthält:

```
  $E2 = % 11100010
+ $34 = % 00110100
-------------------
=$116 = % 100010110
```

Hier kann das Ergebnis nicht mehr durch einen 8-Bit-Wert dargestellt werden, sondern benötigt mit 9 Bit genau ein Bit mehr, als es der Akkumulator zur Verfügung stellt. In diesem Fall wird klar, wozu das Carry-Flag benutzt wird: In ihm wird ein eventueller Übertrag angezeigt. Tritt ein Übertrag auf, wie in unserem letzten Beispiel, wird das Flag gesetzt, im anderen Fall gelöscht. Warum aber wird das Carry-Flag bei jeder Addition mitaddiert, wie oben beschrieben? Nun, dieses Verfahren ist sinnvoll, wenn man Zahlen mit mehr als 8 Bit addieren möchte, z.B. 16-Bit-Zahlen. Als Beispiel sollen die Zahlen $8754 und $52AD addiert werden:

```
  $8754 = % 10000111 01010100
+ $52AD = % 01010010 10101101
-----------------------------
= $DA01 = % 11011010 00000001
```

Hierzu haben wir zunächst die 16-Bit-Werte in Low- und Highbyte zerlegt. Zunächst müssen jetzt die beiden Lowbytes verknüpft werden, wobei offenbar ein Übertrag entsteht. Wenn wir als nächstes dann einfach die Highbytes ohne Berücksichtigung des Übertrags addieren würden, erhielten wir ein falsches Ergebnis. Als Programmierer brauchen wir uns um den Übertrag aber nicht zu kümmern, da dieser ja im Carry-Flag vermerkt wurde und bei der Addition der Highbytes automatisch mitaddiert wurde. Man muß nur aufpassen, daß man vor Beginn der Addition der Lowbytes das Carry-Flag mit dem Befehl CLC, den wir noch kennenlernen, löscht, da sonst eventuell ein falsches Ergebnis herauskommt. Programmtechnisch würde unsere 16-Bit-Addition wie folgt aussehen:

```
CLC       ;Carryflag löschen
LDA #$54  ;Low-Byte Nr.1
ADC #$AD  ;Plus Low-Byte Nr.2
STA $FA   ;merken
LDA #$87  ;High-Byte Nr.1
ADC #$52  ;plus High-Byte Nr.2 plus Übertrag
STA $FB   ;merken
```

Unser 16-Bit-Ergebnis steht nun in den Speicherzellen $FA/$FB zur Verfügung.

Die Subtraktion geschieht analog: Hierbei wird vom Akkuinhalt das adressierte Byte abgezogen. Natürlich besteht die Möglichkeit, daß das Ergebnis kleiner als Null ist. Hier entsteht im Gegensatz zur Addition kein Über-, sondern ein Unterlauf. Dieser wird durch ein gelöschtes Carry-Flag angezeigt, ein gesetztes Flag signalisiert, daß kein Unterlauf aufgetreten ist, sondern sich das Ergebnis im zulässigen Bereich von 0 bis 255 bewegt. Man kann die Wirkung des Befehls

SBC

so formulieren: Akku = Akku – Operand – (1-Carry)

Vor Beginn einer Subtraktion muß daher unbedingt das Carry-Flag gesetzt werden, um Fehler zu vermeiden. Tritt ein Unterlauf auf, wird dieser bei der weiteren Subtraktion berücksichtigt. Es gibt hier 4 verschiedene Fälle zu unterscheiden:

0 – 0 = 0
1 – 0 = 1
0 – 1 = 1 plus Unterlauf
1 – 1 = 0

Hierzu nun ein Beispiel: Der Akku soll den Wert $78 enthalten, von ihm soll die Zahl $92 subtrahiert werden:

  $78 = % 01111000
– $92 = % 10010010
= $E6 = % 11100110

Da ein Unterlauf aufgetreten ist, wird das Carry-Flag gelöscht. Normalerweise müßte unser Ergebnis 120–146= –26 lauten. Unser Ergebnis lautet jedoch $E6 = 230. Die Lösung dieses Rätsels stellt das sogenannte »Zweierkomplement« dar. Um den Wert unseres Ergebnisses zu erhalten, müssen wir nur alle Bits umdrehen und noch 1 hinzuaddieren. Dezimal gesprochen bedeutet dies, daß wir unseren Wert von der Zahl 256 abziehen müssen: 256 – 230 = 26. Binär gesprochen ist jedoch unser Ergebnis $E6 korrekt, was wir überprüfen können, indem wir einfach wieder den Wert $92 hinzuaddieren:

```
  $E6 = % 11100110
+ $92 = % 10010010
------------------
= $78 = % 01111000   plus Übertrag
```

Wichtig ist bei 16-oder-mehr-Bit-Subtraktionen, daß man vor dem Beginn das Carry-Flag setzt, damit der Term 1-Carry Null ergibt und damit ein korrektes Ergebnis zustande kommt. Genau wie bei der Addition muß man hier mit den niederwertigsten Bytes beginnen, z.B. bei einer 16-Bit-Subtraktion zuerst die beiden Lowbytes und dann die Highbytes voneinander abziehen. Ein eventueller Unterlauf der Lowbytes wird damit auf die Highbytes übertragen. Hier nun die Übersicht über die Arithmetik-Befehle:

Befehl: **ADC** — Funktion: A ← (A) + Daten + C

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					69	6D	65	7D	79	75		61	71
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X	X				X	X	Addiere Operand + Carry zum Akku. Ergebnis steht im Akku, Übertrag im Carry.

Befehl: **SBC** — Funktion: A ← (A) – Daten – (1–C)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					E9	ED	E5	FD	FD	F5		E1	F1
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X	X				X	X	Subtrahiere Operand + Carry vom Akku. Ergebnis steht im Akku, Übertrag im Carry.

Bild 1.9: *Die arithmetischen Befehle ADC und SBC*

1.3.5 Die logischen Befehle

Der 6510-Prozessor kennt drei verschiedene logische Befehle. Diese lauten

- AND
- ORA
- EOR

Bei diesen Operationen wird der Inhalt des Akkus bitweise mit dem Inhalt einer Speicherzelle oder einem unmittelbaren Wert verknüpft. Die drei Befehle unterscheiden sich dadurch, daß sie den 4 möglichen Verknüpfungen unterschiedliche Ergebnisse zuweisen:

Verknüpfung	AND	ORA	EOR
0 verknüpft mit 0 =	0	0	0
0 verknüpft mit 1 =	0	1	1
1 verknüpft mit 0 =	0	1	1
1 verknüpft mit 1 =	1	1	0

Während die AND-Verknüpfung nur dann ein »1«-Ergebnis liefert, wenn beide Bits 1 sind, geschieht dies bei der ORA-Verknüpfung, wenn mindestens ein Bit eins ist und bei der EOR-Verknüpfung, wenn genau ein Bit eins ist. Die ersten beiden Operationen kommen auch im Basic vor (AND und OR) –, die Exclusiv-Oder-Funktion, wie die EOR-Verknüpfung auch genannt wird, leider nicht.

1.3.6 Die Zählbefehle

Um die indizierten Adressierungen z.B. bei der Programmierung von Schleifen ausnutzen zu können, stellt uns der Prozessor für beide Indexregister jeweils zwei Befehle zur Verfügung, mit deren Hilfe man diese um eins erhöhen (inkrementieren) oder erniedrigen (dekrementieren) kann:

- INX erhöht X-Register um eins
- DEX erniedrigt X-Register um eins
- ~~INX~~ INY erhöht Y-Register um eins
- ~~DEX~~ DEY erniedrigt Y-Register um eins

Befehl: **AND** Funktion: A ← (A) – Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					29	2D	25	3D	39	35		21	31
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X					X		Bitweise UND-Verknüpfung zwischen Akkuinhalt und Daten.

Befehl: **ORA** Funktion: A ← (A) oder Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					09	0D	05	1D	19	15		01	11
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X					X		Bitweise ODER-Verknüpfung zwischen Akkuinhalt und Daten.

Befehl: **EOR** Funktion: A ← (A) Exclusive-Oder-Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					49	4D	45	5D	59	55		41	51
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X					X		Bitweise EXOR-Verknüpfung zwischen Akkuinhalt und Daten.

Bild 1.10: *Die logischen Befehle AND, ORA und EOR*

Für alle Befehle gilt, daß ein eventueller Über- oder Unterlauf nicht berücksichtigt wird. Wenn also ein Wert von $FF erhöht wird, kommt als Ergebnis eine Null heraus, das Zero-Flag wird gesetzt. Bei einer Verringerung des Wertes $00 entsteht das Ergebnis $FF, wobei das Negativ-Flag gesetzt wird. Bedauerlicherweise existiert kein Befehl, um den Akkuinhalt zu erhöhen oder zu erniedrigen. Dafür existieren jedoch noch zwei weitere Befehle, mit denen man den Inhalt von Speicherzellen verändern kann:

- INC erhöht adressierte Zelle um eins
- DEC erniedrigt adressierte Zelle um eins

Diese beiden Befehle haben zwei interessante Eigenschaften. Zum einen wird ein Wert mit einem Befehl gelesen, verändert und wieder zurückgeschrieben. Alle anderen Befehle können entweder nur eine Speicherzelle lesen oder beschreiben. Zum anderen wird der Akkuinhalt dabei nicht verändert. Wenn man mehr als zwei ineinander verschachtelte Schleifen benötigt, reichen die Indexregister als Zähler nicht mehr aus. In diesem Fall kann man auch eine Speicherzelle als weiteren Zähler benutzen. Das Zero- und Negativ-Flag werden genauso beeinflußt, so daß man das Schleifenende bequem abfragen kann.

1.3.7 Die Verschiebebefehle

Neben den arithmetischen und logischen Befehlen kennt unsere CPU noch eine weitere Gruppe von Befehlen, mit denen Daten verarbeitet werden können. Es handelt sich um die Bit-Schiebebefehle

- ASL schiebt Byte bitweise nach links
- LSR schiebt Byte bitweise nach rechts
- ROL läßt Byte um ein Bit nach links rotieren
- ROR läßt Byte um ein Bit nach rechts rotieren
- Befehl: INX Funktion: X î (X)+1

Befehl: **INX**		Funktion: X ← (X)+1												
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	E8												
Bytes	1												
Takte	2												
Flags	N V B D I Z C X X	Erhöhe den Inhalt des Registers um eins.											

Befehl: **INY**		Funktion: Y ← (Y)+1											
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	C8												
Bytes	1												
Takte	2												
Flags	N V B D I Z C X X	Erhöhe den Inhalt des Registers um eins.											

Bild 1.11: *Die Inkrementierbefehle INX, INY und INC (Fortsetzung nächste Seite)*

Befehl: **INC** Funktion: M ← (M)+1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						EE	E6	FE		F6			
Bytes						3	2	3		2			
Takte						6	5	7		6			

Flags	N	V	B	D	I	Z	C	
	X					X		Erhöhe den Inhalt der Speicherzelle um eins.

Bild 1.11: *Die Inkrementierbefehle INX, INY und INC (Ende)*

Befehl: **DEX** Funktion: Y ← (Y)+1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	CA												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Erniedrige den Inhalt des Registers um eins.

Befehl: **DEY** Funktion: Y ← (Y)+1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	88												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
	X					X		Erniedrige den Inhalt des Registers um eins.

Befehl: **DEC** Funktion: M ← (M)+1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						CE	C6	DE		D6			
Bytes						3	2	3		2			
Takte						6	5	7		6			

Flags	N	V	B	D	I	Z	C	
	X					X		Erniedrige den Inhalt der Speicherzelle um eins.

Bild 1.12: *Die Dekrementierbefehle DEX, DEY und DEC*

Zunächst zu dem Unterschied zwischen den Schiebe- und Rotationsbefehlen. Beim ASL-Befehl wird das hinausgeschobene 7. Bit in das Carry-Flag übertragen. Das freigewordene Bit 0 wird mit dem Wert 0 gefüllt. Genau analog wird beim LSR-Befehl verfahren, nur daß hier das 0. Bit in das Carry-Flag und das Null-Bit in die Position 7 geschoben werden.

Wie der Name »Rotation« schon verrät, wird bei diesen Befehlen anders verfahren: Hier wird das hinausgeschobene Bit zwar auch in das Carry-Flag übertragen, die freigewordene Position wird jedoch nicht mit einem Null-Bit, sondern mit dem vorherigen Inhalt des Carry-Flags aufgefüllt. Wenn man also ein Byte achtmal rotieren läßt, besitzt es den gleichen Inhalt wie vor dem Beginn der Operation.

Anfangs scheint man etwas ratlos zu sein, was man mit diesen Befehlen überhaupt anfangen kann. Ein Beispiel soll daher dieses Rätsel lösen. Wir wollen an dem Wert $46 zunächst den ASL- und dann den LSR-Befehl ausprobieren:

$46 = %01000110 – ASL - %10001100 = $8C, Carry = 0

$46 = %01000110 – LSR - %00100011 = $23, Carry = 0

Man erkennt, daß der Wert durch den Einsatz des ASL-Befehls verdoppelt wurde, während er durch den LSR-Befehl halbiert wurde. Dies ist ganz logisch, da ja im Binärsystem die folgende Stelle immer den doppelten Wert innehat.

Man hat eine einfache Möglichkeit gefunden, Multiplikationen und Divisionen mit Faktoren durchzuführen, die durch zwei teilbar sind. Wenn man z.B. den ASL-Befehl dreimal ausführen würde, wäre das Ergebnis achtmal so groß (8=2*2*2) wie vorher.

Durch einfache Addition von mehreren Multiplikationsergebnissen kann man jeden Faktor darstellen. Will man eine Zahl »Z« z.B. mit dem Faktor 15 multiplizieren, kann man die Rechnung

Z * 15 = Z*2*2*2 + Z*2*2 + Z*2 + Z

durchführen. Die Rotationsbefehle werden bei Verschiebeoperationen von mehr als 8 Bit wirksam, da das Carry-Flag als Übertrag fungiert. Wir wollen z.B. die Zahl $00F3 mal vier nehmen. Wir müssen die 16 Bit zweimal um ein Bit nach links rotieren lassen. Dafür muß beim Lowbyte der ASL- und beim Highbyte der ROL-Befehl eingesetzt werden, um einen eventuellen Übertrag zu erfassen:

1. Schritt: \$F3 = %11110011 – ASL – %11100110 = \$E6 , Carry = 1
\$00 = %00000000 – ROL – %00000001 = \$01 , Carry = 0
2. Schritt: \$E6 = %11100110 – ASL – %11001100 = \$D8 , Carry = 1
\$01 = %00000001 – ROL – %00000011 = \$03 , Carry = 0

Unser Ergebnis lautet also \$03D8 = 4 * \$F6.

Wahlweise können Akkumulator oder Speicherzellen adressiert werden. Im zweiten Fall wird wie beim INC- oder DEC-Befehl gelesen, verändert und geschrieben, ohne den Akkuinhalt zu verändern.

Befehl: **ASL** Funktion: C ← 7 6 5 4 3 2 1 0 ← 0

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX		0A				0E	06	1E		16			
Bytes		1				3	2	3		2			
Takte		2				6	5	7		6			

Flags	N	V	B	D	I	Z	C	
	X					X	X	Schiebe Byte um eine Bitstelle nach links.

Befehl: **LSR** Funktion: 0 → 7 6 5 4 3 2 1 0 → 0

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX		4A				4E	46	5E		56			
Bytes		1				3	2	3		2			
Takte		2				6	5	7		6			

Flags	N	V	B	D	I	Z	C	
	X					X	X	Schiebe Byte um eine Bitstelle nach links.

Bild 1.13: *Die Schiebebefehle ASL und LSR*

Befehl: **ROL** Funktion: C ← 7 6 5 4 3 2 1 0 ← 0

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX		2A				2E	26	3E		36			
Bytes		1				3	2	3		2			
Takte		2				6	5	7		6			

Flags	N	V	B	D	I	Z	C	
	X					X	X	Rotiere Byte um eine Bitstelle nach links.

Bild 1.14: *Die Rotationsbefehle ROL und ROR (Fortsetzung nächste Seite)*

Befehl: **ROR**					Funktion: C → 7 6 5 4 3 2 1 0 → 0								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX		6A				6E	66	7E		76			
Bytes		1				3	2	3		2			
Takte		2				6	5	7		6			
Flags	N V B D I Z C				Schiebe Byte um eine Bitstelle nach links.								
	X X X												

Bild 1.14: *Die Rotationsbefehle ROL und ROR (Ende)*

1.3.8 Die Vergleichsbefehle

Viele Probleme lassen sich gerade deswegen so gut mit Hilfe der Datenverarbeitung lösen, weil sehr viele Vergleichsoperationen schnell ausgeführt werden können. Das Paradebeispiel hierfür stellt das Sortieren von Namen oder anderen Daten im Rahmen einer Dateiverwaltung dar. Um eine riesige Liste zu sortieren, muß man jeweils zwei Datensätze vergleichen und eventuell vertauschen. Für Computer wurden mehrere Sortieralgorithmen entwickelt, z.B. Quicksort. Im Kapitel »Variablenverwaltung« werden Sie dann den Bubblesort-Algorithmus kennenlernen. Auf jeden Fall wäre man ohne Vergleichsbefehle hilflos. Diese existieren für den Akkumulator und die beiden Indexregister:

- CMP Vergleiche mit Inhalt des Akkus
- CPX Vergleiche mit Inhalt des X-Registers
- CPY Vergleiche mit Inhalt des Y-Registers

Intern subtrahiert der Prozessor den Operanden von dem angesprochenen Register und setzt die Flags Carry-, Zero- und Negative je nach Ergebnis. Dabei werden weder Registerinhalt noch der Operand verändert. Tritt ein Unterlauf ein, wird das Carry-Flag gelöscht, sonst wird es gesetzt. Falls das Ergebnis Null ist, wird das Zero-Flag gesetzt, sonst gelöscht. Das Negative-Flag wird gesetzt, wenn das Ergebnis größer als 127 ist. Für die praktische Anwendung kann man daher folgenden Zusammenhang feststellen:

C = 1 bedeutet größer oder gleich
Z = 1 bedeutet gleich
C = 0 bedeutet kleiner

Um entscheiden zu können, ob der Registerinhalt größer als der Operand ist, müssen daher Carry- und Zero-Flag überprüft werden:

Z = 0 und C =1 bedeutet größer

Das Negative-Flag kann im allgemeinen nicht als Entscheidungsgrundlage benutzt werden.

1.3.9 Die Befehle zur bedingten Verzweigung

Um die Zustände der Flags für unser Programm ausnutzen zu können, gibt es Befehle, mit deren Hilfe man Entscheidungen treffen kann. Für die Flags C, Z, N und V existieren jeweils zwei Sprungbefehle mit entgegengesetzter Wirkung. Während der eine immer bei gesetztem Flag springt, tut dies der andere bei gelöschtem. Damit können wir z.B. auf die Vergleichsbefehle reagieren. Da unser Prozessor wissen muß, wohin er springen soll, müßte normalerweise nach dem Befehlscode ein 2-Byte-Operand folgen, der die Zieladresse im Format Low- und Highbyte angibt. Der Nachteil dieser Adressierung wäre jedoch darin zu sehen, daß die Sprungadresse bei jeder Speicherverschiebung des Programms mitgeändert werden müßte.

Befehl: **CMP** Funktion: N, Z, C ← (A) –Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					C9	CD	C5	DD	D9	D5		C1	D1
Bytes					2	3	2	3	3	2		2	2
Takte					2	4	3	*4	*4	4		6	*5

Flags	N	V	B	D	I	Z	C	
	X					X	X	Operand wird vom Register abgezogen, die Flags C, Z und N entsprechend gesetzt.

Befehl: **CPX** Funktion: N, Z, C ← (X) –Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					E0	EC	E4						
Bytes					2	3	2						
Takte					2	4	3						

Flags	N	V	B	D	I	Z	C	
	X					X	X	Operand wird vom Register abgezogen, die Flags C, Z und N entsprechend gesetzt.

Befehl: **CPY** Funktion: N, Z, C ← (Y) –Daten

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					C0	CC	C4						
Bytes					2	3	2						
Takte					2	4	3						

Flags	N	V	B	D	I	Z	C	
	X					X	X	Operand wird vom Register abgezogen, die Flags C, Z und N entsprechend gesetzt.

Bild 1.15: *Die Vergleichsbefehle CMP, CPX und CPY*

Da man in der Praxis sehr viele Sprünge dieser Art benötigt, hat man sich die relative Adressierung einfallen lassen. Hierbei folgt nach dem Befehlscode ein Offset, der den Abstand vom augenblicklichen Stand des Programmzählers angibt. Diese Methode erlaubt ein beliebiges Verschieben des Programms im Speicher, da sich die relative Sprungadresse ja nicht ändert. Außerdem wird hierbei nur ein Byte für den Offset benötigt, wodurch Speicherplatz und Rechenzeit gespart werden. Der Nachteil dieser Methode ist darin zu sehen, daß man nicht beliebig weit, sondern nur im Rahmen von 256 Byte springen kann. Um auch rückwärts springen zu können, werden Offsets, die größer als 127 sind, als Zweierkomplement von negativen Zahlen interpretiert. Damit kann man um 127 Byte vorwärts und um 128 Byte rückwärts springen. Aus der Tabelle in Bild 1.16 können Sie den Zusammenhang zwischen den Offsets und den zu springenden Bytes ersehen. Wenn Sie z.B. um 25 Byte nach vorne springen wollen, lesen Sie aus der linken Randspalte das obere Nibble des Offsets und aus der obersten Zeile das untere Nibble ab. Hier beträgt der Offset also $19. In der Praxis wird uns die Offset-Berechnung vom Assembler abgenommen, da dieser mit Labels (Namen der Sprungadressen) arbeitet.

Die Befehle zur Verzweigung bei gesetztem bzw. gelöschtem Zero-Flag heißen

- BEQ bei gesetztem Zero-Flag
- BNE bei gelöschtem Zero-Flag

Wollen Sie z.B. den Inhalt des Akkumulators mit dem der Speicherstelle $FA vergleichen und bei Gleichheit verzweigen, genügt die Befehlsfolge

```
CMP $FA         ;Vergleiche Akku mit Zelle $FA
BEQ SPRUNG      ;Wennnn gleich, dann Sprung
```

Die entsprechenden Befehle für das Carry-Flag lauten

- BCS bei gesetztem Carry-Flag
- BCC bei gelöschtem Carry-Flag

Wie wir im letzten Abschnitt gesehen haben, kann man nach einem Vergleich nicht unmittelbar auf Grund des Carry-Flags zwischen »größer« und »gleich« unterscheiden. Wir wollen diesmal verzweigen, wenn der Akku einen größeren Inhalt als die Speicherzelle aufweist:

```
    CMP $FA    ;Vergleiche Akku mit Zelle $FA
    BCC W1     ;Wenn kleiner, weiter
    BEQ W1     ;Wenn gleich, weiter
    BCS SPRUNG;Grösser, also Sprung
W1  ...
```

Hauptsächlich wird man seine Entscheidungen auf Grund des Carry- und Zero-Flags treffen. Für die Verzweigung mit Hilfe des Negative- und Overflow-Flags stehen weitere 4 Befehle zur Verfügung (siehe Abbildung 1.18 und 1.19):

Offset für relative Vorwärtssprünge

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Offset für relative Rückwärtssprünge

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Bild 1.15: *Offsets bei relativen Sprüngen*

- BMI verzweigt bei gesetztem Negative-Flag
- BPL verzweigt bei gelöschtem Negative-Flag
- BVS verzweigt bei gesetztem Overflow-Flag
- BVC verzweigt bei gelöschtem Overflow-Flag

Im Gegensatz zum C64 hat das Overflow-Flag in der Floppy 1541 eine entscheidende Bedeutung: Es zeigt dort an, wann ein Byte auf die Diskette geschrieben wurde bzw. wann 1 gelesen wurde.

Befehl: **BEQ**	Funktion: Wenn Z=1 ist, springe

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			F0										
Bytes			2										
Takte			#2										

Flags N V B D I Z C	Ist Z=1, springe. Der Sprungbereich reicht von –128 bis +127 Byte.

Befehl: **BNE**	Funktion: Wenn Z=0 ist, springe

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			D0										
Bytes			2										
Takte			#2										

Flags N V B D I Z C	Ist Z=0, springe. Der Sprungbereich reicht von –128 bis +127 Byte.

Bild 1.17: *Die bedingten Sprungbefehle BEQ und BNE*

Befehl: **BCS**	Funktion: Wenn C=1 ist, springe

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			B0										
Bytes			2										
Takte			#2										

Flags N V B D I Z C	Ist C=1, springe. Der Sprungbereich reicht von –128 bis +127 Byte.

Befehl: **BCC**	Funktion: Wenn C=0 ist, springe

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			90										
Bytes			2										
Takte			#2										

Flags N V B D I Z C	Ist C=0, springe. Der Sprungbereich reicht von –128 bis +127 Byte.

Bild 1.18: *Die bedingten Sprungbefehle BCS, BCC und BMI* *(Fortsetzung nächste Seite)*

Befehl: **BMI**				Funktion: Wenn N=1 ist, springe									
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			30										
Bytes			2										
Takte			#2										
Flags N V B D I Z C				Ist N=1, springe. Der Sprungbereich reicht von –128 bis +127 Byte.									

Bild 1.18: *Die bedingten Sprungbefehle BCS, BCC und BMI (Ende)*

Befehl: **BPL**				Funktion: Wenn N=0, ist, springe									
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			10										
Bytes			2										
Takte			#2										
Flags N V B D I Z C				Ist N=0, springe. Der Sprungbereich reicht von –128 bis +127 Byte.									

Befehl: **BVS**				Funktion: Wenn V=1 ist, springe									
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			70										
Bytes			2										
Takte			#2										
Flags N V B D I Z C				Ist V=1, springe. Der Sprungbereich reicht von –128 bis +127 Byte.									

Befehl: **BVC**				Funktion: Wenn V=0 ist, springe									
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			50										
Bytes			2										
Takte			#2										
Flags N V B D I Z C				Ist V=0, springe. Der Sprungbereich reicht von –128 bis +127 Byte.									

Bild 1.19: *Die bedingten Sprungbefehle BPL, BVS und BVC*

1.3.10 Die Befehle zur Beeinflussung der Flags

In den vorhergehenden Abschnitten haben wir gesehen, wie die Flags durch bestimmte Maschinenbefehle beeinflußt werden. Vier dieser Flags können jedoch auch »per Hand« gesetzt oder gelöscht werden. So ist es ja auch z.B. notwendig, vor Additionen das Carry-Flag zu löschen und vor einer Subtraktion zu setzen. Hier nun die sieben 1-Byte-Befehle (das Overflow-Flag kann man nur löschen, nicht aber setzen):

- CLC löscht Carry-Flag
- SEC setzt Carry-Flag
- CLD löscht Dezimal-Flag
- SED setzt Dezimal-Flag
- CLI löscht Interrupt-Flag
- SEI setzt Interrupt-Flag
- CLV löscht Overflow-Flag

In den Kapiteln »Interruptprogrammierung« und »Variablenverwaltung« werden Sie sehen, wie man das Interrupt- und Dezimal-Flag sinnvoll einsetzen kann. Falls Sie auch am Overflow-Flag interessiert sind, kann ich Ihnen das Buch »Die Floppy 1541«, erschienen im Markt & Technik Verlag, empfehlen, das sich ausführlich mit der Programmierung der Floppystation und damit auch mit dem Overflow-Flag befaßt.

Befehl: **CLC**					Funktion: C ← 0									
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
HEX	18													
Bytes	1													
Takte	2													
Flags	N	V	B	D	I	Z	C	Das Carry-Flag wird auf 0 gesetzt.						
							0							

Bild 1.20: *Der CLC-Befehl zum Löschen des Carry-Flags*

Befehl: **SEC**	Funktion: C ← 1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	38												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
							1	Das Carry-Flag wird auf 1 gesetzt.

Befehl: **CLD**	Funktion: D ← 0

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	D8												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
				0				Das Dezimal-Flag wird auf 0 gesetzt.

Befehl: **SED**	Funktion: D ← 1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	F8												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
				1				Das Dezimal-Flag wird auf 1 gesetzt.

Bild 1.21: *Die Flag-Befehle SEC, CLD und SED*

Befehl: **CLI**	Funktion: I ← 0

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	58												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
					0			Das Interrupt-Flag wird auf 0 gesetzt.

Bild 1.22: *Die Flag-Befehle CLI, SEI und CLV (Fortsetzung nächste Seite)*

Befehl: **SEI**					Funktion: I ← 1								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	78												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
					1			Das Interrupt-Flag wird auf 1 gesetzt.

Befehl: **CLV**					Funktion: V ← 0								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	B8												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	
		0						Das Overflow-Flag wird auf 0 gesetzt.

Bild 1.22: *Die Flag-Befehle CLI, SEI und CLV (Ende)*

1.3.11 Die unbedingten Sprungbefehle

Genauso wie es in Basic mit dem GOTO-Befehl möglich ist, das Programm an einer anderen Stelle fortzuführen, kann man dies auch in der Maschinensprache realisieren. Der Befehl

JMP

führt dazu, daß der Programmzähler mit der als Operand angegebenen Adresse geladen und der nächste Befehlscode dann aus dieser Adresse geholt wird. Neben dem absoluten ist aber auch ein indirekter Sprung möglich. Von dieser Möglichkeit machen Basic-Interpreter und Betriebssystem reichlich Gebrauch.

Befehl: **JMP**					Funktion: PC ← ADR								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX				6C		4C							
Bytes				3		4							
Takte				3		5							

Flags	N	V	B	D	I	Z	C	
								Der PC wird mit der angegebenen Adresse geladen.

Bild 1.23: *Der unbedingte Sprungbefehl JMP*

Sie holen sich die Sprungadressen aus dem RAM und bieten damit die Möglichkeit, modifiziert zu werden. Von dieser Möglichkeit machen alle professionellen Basic-Erweiterungen Gebrauch.

1.3.12 Die Unterprogrammbefehle

Ähnlich wie in Basic, wo man Unterprogramme mit dem Befehl GOSUB aufrufen und mit RETURN beenden kann, gibt es auch in Maschinensprache die Möglichkeit, diese Programmiertechnik zu nutzen. Die Befehle hierfür lauten

- JSR Aufruf eines Unterprogramms
- RTS Beendigung eines Unterprogramms

Wenn der Prozessor auf den JSR-Befehl trifft, nimmt er den augenblicklichen Stand des Programmzählers plus 2 und zerlegt diesen zunächst in Low- und Highbyte. Anschließend werden diese beiden in umgekehrter Reihenfolge (d.h. zuerst das Highbyte) auf den Stack geschoben. Dann wird der Operand hinter dem Befehlscode in den Programmzähler geladen, wie es auch beim JMP-Befehl geschieht. Nach der Ausführung des Unterprogramms wird der umgekehrte Weg gegangen: Die Rücksprungadresse wird vom Stack geholt und um 1 erhöht. Damit wird der Programmzähler geladen und das Programm wird hinter dem JSR-Befehl fortgeführt. Der Grund für die Erhöhung des Programmzählers um 2 (beim Aufruf) und nochmals um 1 (beim Rücksprung) liegt auf der Hand: Da der JSR-Befehl 3 Byte lang ist, müssen diese natürlich nach der Abarbeitung des Unterprogramms übersprungen werden. Das folgende Beispiel soll die Vorgänge verdeutlichen:

```
$1234    JSR $C000   S = $FF
                     $01FF = $12, S = S-1
                     $01FE = $36, S = S-1
```

Nachdem das Unterprogramm ausgeführt wurde, können wir dieses wieder beenden:

```
$XXXX    RTS         S = $FD
                     S = S+1, PCL = ($01FE) = $36+1
                     S = S+1, PCH = ($01FF) = $12
```

Unser Hauptprogramm wird an der Adresse $1237 fortgesetzt. Vielleicht fragen Sie sich, warum der Stackpointer nach dem Stapeln der Rücksprungadresse nochmals erniedrigt wird. Nun, falls weitere Daten innerhalb des Unterprogramms gestapelt werden sollen, muß der Stackpointer auf die nächste freie Adresse zeigen, da sonst das Lowbyte der Rücksprungadresse überschrieben würde. Dies würde natürlich zu einer völlig falschen Rücksprungadresse führen. Genauso schlimm ist es, wenn Sie innerhalb des Unterprogramms Werte auf den Stack speichern und diese vor der Beendigung nicht wieder herunterholen. Die einzige Möglichkeit besteht darin, den Stackpointer mit Hilfe des TXS-Befehls wieder auf die richtige Adresse zu setzen. Für Maschinenspracheverhältnisse benötigen die Unterprogrammbefehle – genau wie im Basic – sehr viel Zeit zur Ausführung.

Befehl: **JSR**					Funktion: PC ← ADR, Stack ← (PC)+2								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						20							
Bytes						3							
Takte						6							
Flags	N V B D I Z C					Das Programm wird bei ADR fortgesetzt, der PC+2 auf den Stack gebracht.							

Befehl: **RTS**					Funktion: PC ← (Stack), S ← (S)+1, PC ← (PC)+1								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						60							
Bytes						1							
Takte						6							
Flags	N V B D I Z C					Der PC wird vom Stapel geholt und erhöht. Der S-Pointer wird neu gesetzt.							

Bild 1.24: *Die Unterprogramm-Befehle JSR und RTS*

1.3.13 Die Stackbefehle

Wie schon im letzten Abschnitt angedeutet, besteht die Möglichkeit, den Stapel als Zwischenspeicher für Daten zu verwenden. Es besteht genauer gesagt die Möglichkeit, den Akkumulator sowie das Statusregister abzuspeichern und natürlich auch wieder herunterzuholen. Die vier Befehle hierfür lauten:

- PHA Speichere den Inhalt des Akkus auf dem Stapel
- PLA Hole Byte vom Stapel in den Akkumulator
- PHP Speichere den Inhalt des Statusregisters auf dem Stapel
- PLP Hole Byte vom Stapel in das Statusregister

Der Stapelzeiger wird automatisch korrigiert.

Befehl: **PHA** — Funktion: Stack ← (A), S ← (S)–1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	48												
Bytes	1												
Takte	3												

Flags	N	V	B	D	I	Z	C	
								Bringe den Akku auf den Stapel. Erniedrige den Stapelzeiger.

Befehl: **PLA** — Funktion: S ← (S)+1, A ← (Stack)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	68												
Bytes	1												
Takte	4												

Flags	N	V	B	D	I	Z	C	
								Erhöhe den Stapelzeiger. Lade den Akku mit dem obersten Wert des Stapels.

Befehl: **PHP** — Funktion: Stack ← (P), S ← (S)–1

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	08												
Bytes	1												
Takte	3												

Flags	N	V	B	D	I	Z	C	
								Der Statusregisterinhalt wird gestapelt. Er wird dabei nicht verändert.

Bild 1.25: *Die Stackbefehle PHA, PLA und PHP*

Befehl: **RTI** — Funktion: S ← (S)+1, P ← (Stack)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	28												
Bytes	1												
Takte	4												

Flags	N	V	B	D	I	Z	C	
	X	X	X	X	X	X	X	Das Statusregister wird mit der Spitze des Stacks geladen.

Bild 1.26: *Der Stackbefehl PLP*

1.3.14 Die Interruptbefehle

Da diese zwei Befehle im Kapitel »Interruptprogrammierung« sehr ausführlich besprochen werden, hier nur die tabellarische Auflistung:

- RTI Beendet Interruptroutine
- BRK Befehl für Software-Interrupt

Befehl: **RTI**	Funktion: P ← (Stack), PC ← (Stack), S ← (S)+3												
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	40												
Bytes	1												
Takte	6												

Flags	N	V	B	D	I	Z	C	
	X	X	X	X	X	X	X	Es wird der ursprüngliche Zustand des Status + PC vor dem Interrupt erzeugt.

Bild 1.27: *Der Interruptbefehl RTI*

Befehl: **BRK**	Funktion: Stack ← (PC)+2, Stack ← (P), PC ← ($FFFE, $FFFF)												
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	00												
Bytes	1												
Takte	7												

Flags	N	V	B	D	I	Z	C	
			*		1			PC und Status werden gerettet. In den PC wird die Adr in $FFFE geholt.

Bild 1.28: *Der Interruptbefehl BRK*

1.3.15 Die Sonderbefehle

Die CPU 6510 bietet noch zwei Befehle an, die sich in die bislang besprochenen Gruppen nicht einordnen lassen. Dies sind

- BIT und
- NOP

Der BIT-Befehl stellt eine AND-Verknüpfung zwischen dem Inhalt des Akkus und der adressierten Speicherzelle her. Ist das Ergebnis Null, wird das Zero-Flag gesetzt, sonst wird es gelöscht. Zusätzlich wird das 6. Bit der adressierten Speicherzelle in das Overflow-Flag und das 7. Bit in das Negative-Flag übertragen. Der BIT-Befehl stellt die einzige Möglichkeit dar, mit

der das 6. Bit einer Speicherzelle abgefragt werden kann. Danach kann mit den Befehlen BMI, BPL, BVC und BVS je nach Flaggenzustand verzweigt werden. Der Akkuinhalt bleibt unverändert. Dieser Befehl wird auch gerne für einen kleinen Trick verwendet: Wenn man zwischen zwei Maschinenbefehlen ein Byte $2C (Code für Bit-Befehl) einfügt, wird der zweite Befehl übersprungen, falls es sich um einen 2-Byte-Befehl handelt. Man braucht dann keinen Sprungbefehl einzubauen.

Der NOP-Befehl ist hingegen äußerst primitiver Natur: er tut 2 Taktzyklen überhaupt nichts! Er findet Verwendung als Platzhalter und Verzögerer in Schleifen.

Befehl: **BIT**						Funktion: Z ← (A) und (M), N ← (M7), V ← (M6)								
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						2C	24						
Bytes						3	2						
Takte						4	3						
Flags	N V B D I Z C 7 6 X					Die Inhalte von Akku und M werden durch die UND-Funktion verknüpft. Das Z-Flag wird abhängig vom Ergebnis gesetzt. Der Akku bleibt unverändert. Bit 7 und 6 von M werden in das N- bzw. V-Flag kopiert.							

Befehl: **NOP**						Funktion: keine							
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX	EA												
Bytes	1												
Takte	2												
Flags	N V B D I Z C					Der Befehl tut zwei Taktzyklen absolut nichts ! ! !							

Bild 1.29: *Die Sonderbefehle BIT und NOP*

1.3.16 Die illegalen Opcodes

Von den 256 möglichen Befehlscodes sind nur 151 für legale Maschinenbefehle genutzt worden. Die übrigen 104 Codes sind vom Hersteller nicht etwa mit NOPs unterlegt worden, sondern führen teilweise recht sinnvolle Operationen durch. Man nennt sie »illegale Opcodes«, da sie herstellungsbedingt nicht auf allen Prozessoren funktionieren. Früher wurden die Opcodes gern von Software-Firmen für Kopierschutz-Zwecke herangezogen, da es keinen Monitor gab, der die Befehle disassemblieren konnte. Es folgte jedoch ein Sturm der Entrüstung der Käufer, auf deren Rechner die Programme nicht liefen. Heute ist man von dieser Methode wieder abgerückt, zumal gute Monitore, wie z.B. der diesem Buch beiliegende

SMON, die illegalen Codes durchaus verarbeiten können. Sie sollten diese Codes auf keinen-Fall in Programme einbauen, die Sie einmal veröffentlichen wollen, sondern nur in solche, die für Ihren eigenen Bedarf geschrieben sind. So wurde auch in diesem Buch auf illegale Opcodes verzichtet, obwohl man an einigen Stellen durch ihre Verwendung Platz und Rechenzeit hätte sparen können. Man kann die Codes grob in drei Gruppen einteilen:

- Die Mehrfachausführer
- Die Nichtstuer
- Die Killercodes

veranlassen, mehrere legale Befehle hintereinander auszuführen. So bewirkt z.B. der Befehl LAX ein Laden des Akkumulators und ein anschließendes Transferieren in das X-Register, man könnte sagen, die Befehle LDA und TAX werden hintereinander ausgeführt. Bei den Nichtstuern handelt es sich um 1,- 2- und 3-Byte-NOPs. Bei den zwei letzten Befehlen werden die beiden Bytes (ein Byte bei 2-Byte-NOPs) hinter dem Befehlscode einfach übersprungen. Bitte beachten Sie, daß sich die Ausführungszeiten der 2-Byte-Befehle im Bereich von zwei bis sechs Taktzyklen bewegen, d.h. ein Befehl, der genau dieselbe Wirkung erzielt, benötigt dreimal soviel Zeit wie ein anderer!

Die Killercodes gehören zu den unschönsten Erscheinungen beim Programmieren. Sie bewirken nämlich einen totalen Absturz des Prozessors, der nur mit einem Reset beendet werden kann. Man kann davon ausgehen, daß das Programm mehr oder weniger zerstört wurde.

Befehl: **A11** Funktion: M ← (Register) und ($11)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX								9C	9E				
Bytes								3	3				
Takte								5	5				

Flags	N	V	B	D	I	Z	C	
	X					X		Verknüpfe Register mit Speicher durch UND, speichere Ergebnis.

Befehl: **AAX** Funktion: M ← (A) und (X)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX					8B	8F	87				97	83	
Bytes					2	3	2				2	2	
Takte					2	4	3				4	6	

Flags	N	V	B	D	I	Z	C	
	X					X		Verknüpfe Akku und X-Reg mit UND. Der Akkuinhalt wird abgespeichert.

Bild 1.30: *Die illegalen Opcodes A11, AAX und ASR (Fortsetzung nächste Seite)*

Befehl: **ASR** Funktion: A ← (A) und Daten, A ← (A) Shift Right

Adress.	Imp A Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			6B								
Bytes			2								
Takte			2								

Flags	N	V	B	D	I	Z	C	
	0					X	X	Es wird die Befehlsfolge AND – LSR ausgeführt.

Bild 1.30: *Die illegalen Opcodes A11, AAX und ASR (Ende)*

Befehl: **ARR** Funktion: A ← (A) und (D), A ← (A) Rotate Right

Adress.	Imp A Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			4B				9E				
Bytes			2				3				
Takte			2				5				

Flags	N	V	B	D	I	Z	C	
	X					X	X	Es wird die Befehlsfolge AND – ROR ausgeführt.

Befehl: **AXS** Funktion: A ← (A) und (X), A ← (A) – Daten

Adress.	Imp A Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX			CB								
Bytes			2								
Takte			2								

Flags	N	V	B	D	I	Z	C	
	X	X				X	X	Akku wird mit X-Reg durch UND verknüpft. Davon wird der Wert subtrahiert.

Befehl: **DCP** Funktion: M ← (M) –1, N, Z, C ← (A) –Daten

Adress.	Imp A Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX				CF	C7	DF	DB	D7		C3	D3
Bytes				3	2	3	3	2		2	2
Takte				6	5	7	7	6		8	8

Flags	N	V	B	D	I	Z	C	
	X					X	X	Es wird die Befehlsfolge DEC – CMP ausgeführt.

Bild 1.31: *Die illegalen Opcodes ARR, AXS und DCP*

Befehl: **DOP**												
Funktion: Tut nichts (Double NOP)												
Die Adressierung ist implizit. Folgende Befehle existieren:												
HEX	04	14	34	44	54	64	74	D4	F4	80	89	93
Bytes	2	2	2	2	2	2	2	2	2	2	2	2
Takte	3	4	4	3	4	3	4	4	4	2	2	6
Flags	N V B D I Z C	Wirkung wie beim NOP, das folgende Byte wird jedoch übersprungen.										

Befehl: **ISC**	Funktion: M ← (M) –1, N, Z, C, (A) ← (A) – (M)												
Adress.	Imp	A	Rel	()	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	(),Y
HEX						EF	E7	FF	FB	F7		E3	F3
Bytes						3	2	3	3	2		2	2
Takte						6	5	7	7	6		8	8
Flags	N V B D I Z C X X X X	Es wird die Befehlsfolge INC – SBC ausgeführt.											

Befehl: **LAR**	Funktion: A, X, S ← (S) und (M)												
Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX									BB				
Bytes									3				
Takte									4				
Flags	N V B D I Z C X X	Es wird die Befehlsfolge LDA – AND – TAX – TXS ausgeführt.											

Bild 1.32: *Die illegalen Opcodes DOP, ICS und LAR*

Befehl: **KIL**	Funktion: Absturz des Prozessors											
Die Adressierung ist implizit. Folgende Befehle existieren:												
HEX	02	12	22	32	42	52	62	72	92	B2	D2	F2
Die CPU stürzt komplett ab. Nur ein Reset kann helfen.												

Bild 1.33: *Die illegalen Opcodes KIL, LAX, NOP und TOP (Fortsetzung nächste Seite)*

Befehl: **LAX** Funktion: A, X ← (M)

Adress.	Imp	A	Rel	()	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	(),Y
HEX						AF	A7		BF		B7	A3	B3
Bytes						3	2		3		2	2	2
Takte						4	3		4		4	6	5

Flags	N	V	B	D	I	Z	C
	X					X	

Es wird die Befehlsfolge LDA – TAX ausgeführt.

Befehl: **NOP** Funktion: Nichts

Die Adressierung ist implizit. Folgende Befehle existieren:

HEX 1A 3A 5A 7A DA FA

Die Opcodes haben dieselben Eigenschaften wie das normale NOP.

Befehl: **TOP** Funktion: Tut nichts (Triple NOP)

Die Adressierung ist implizit. Folgende Befehle existieren:

HEX 0C 1C 3C 5C 7C DC FC

Bis auf den Code $0C (4 Taktzyklen) benötigen alle 5 Taktzyklen.
Wirkung wie bei NOP, jedoch werden zwei Byte übersprungen.

Bild 1.33: *Die illegalen Opcodes KIL, LAX, NOP und TOP (Ende)*

Befehl: **RLA** Funktion: (M) ← ROL, A ← (A) und (M), M ← (A)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						2F	27	3F	3B	37		23	33
Bytes						3	2	3	3	2		2	2
Takte						6	5	7	7	6		8	8

Flags	N	V	B	D	I	Z	C
	X					X	X

Es wird die Befehlsfolge ROL – AND – STA ausgeführt.

Bild 1.34: *Die illegalen Opcodes RLA, RRA und SLO (Fortsetzung nächste Seite)*

Befehl: **RRA** Funktion: (M) ← ROR, A ← (A) + (M)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						6F	67	7F	7B	77		63	73
Bytes						3	2	3	3	2		2	2
Takte						6	5	7	7	6		8	8

Flags	N	V	B	D	I	Z	C	
	X	X				X	X	Es wird die Befehlsfolge ROR – ADC ausgeführt.

Befehl: **SLO** Funktion: (M) ← ASL, A ← (A) oder (M)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						0F	07	1F	1B	17		03	13
Bytes						3	2	3	3	2		2	2
Takte						6	5	7	7	6		8	8

Flags	N	V	B	D	I	Z	C	
	X					X	X	Es wird die Befehlsfolge ASL – ORA ausgeführt.

Bild 1.34: *Die illegalen Opcodes RLA, RRA und SLO (Ende)*

Befehl: **SRE** Funktion: (M) ← LSR, A ← (A) exor (M)

Adress.	Imp	A	Rel	( )	#	Abs	ZP	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
HEX						4F	47	5F	5B	57		43	53
Bytes						3	2	3	3	2		2	2
Takte						6	5	7	7	6		8	8

Flags	N	V	B	D	I	Z	C	
	X					X	X	Es wird die Befehlsfolge LSR – EXOR ausgeführt.

Bild 1.35: *Der illegale Opcode SRE*

1.4 Das Assembler-Entwicklungssystem

Nachdem wir nun die umfangreiche Einführung in die Maschinensprache hinter uns gebracht haben, wollen wir diese Kenntnisse natürlich in die Praxis umsetzen. Früher, als es weder Assembler noch Monitor für den C64 gab, mußte das Maschinenprogramm auf dem Papier entworfen und dann die Werte in den Speicher geschrieben werden, was mittels POKE-Befehl geschah. Man kann sich leicht ausrechnen, daß diese Methode nicht nur sehr fehleranfällig war, sondern auch bei Korrekturen unmenschliche Arbeit abverlangte.

Heute gibt es Werkzeuge, mit denen die Erstellung von Maschinenprogrammen fast so einfach vor sich geht, wie man es vom Basic-Interpreter gewohnt ist. Auf der Diskette zum Buch befindet sich ein Entwicklungssystem, das ohne Übertreibung zu den besten für den C64 gehört und professionellen Anforderungen genügt:

- Der Hypra-Ass-Plus-Makroassembler mit dazugehörigem Reassembler
- Der Maschinensprachemonitor SMON Plus mit integriertem Einzelschrittsimulator und Diskettenmonitor

Während der Assembler dazu dient, neue Programme zu entwickeln, kann man diese mit dem Monitor dann direkt im Speicher sehr komfortabel editieren. Um die sehr schwierige Fehlersuche zu erleichtern, kann man mit Hilfe des Einzelschrittsimulators ein Maschinenprogramm Schritt für Schritt ablaufen lassen. Nach jedem Schritt wird die Wirkung auf die Flags und die einzelnen Register genau angezeigt. Der SMON ist sogar in der Lage, die im vorigen Abschnitt besprochenen illegalen Opcodes zu verarbeiten! Falls man ein Programm grundsätzlich ändern möchte, ist der Weg über den Monitor zu umständlich. Hier ist der Einsatz des Reassemblers angesagt, der ein Maschinenprogramm wieder in einen Quelltext zurückverwandelt, der leicht mit dem Hypra-Ass editiert werden kann.

Man kann zusammenfassend sagen, daß Sie mit diesem Entwicklungssystem wohl optimal ausgestattet sind und jede noch so schwierige Situation meistern können.

1.4.1 Der Hypra-Ass-Plus-Makroassembler

Hypra-Ass ist ein von Gerd Möllmann entwickelter 3-Pass-Makroassembler mit integriertem Editor und Reassembler. Er wird mit

```
LOAD "HYPRA-ASS PLUS",8
```

geladen und mit RUN gestartet. Nach dem Start meldet sich Hypra-Ass mit »Break in 0« und »Ready«. Bis auf die Befehle LET und FOR können weiterhin alle Basic-Befehle verwendet werden. Der Befehl RUN dient zum Start der Assemblierung.

1.4.1.1 Der Quelltext

Der Quelltext wird vom Hypra-Ass-Editor in Basic-Programmzeilen abgelegt. Soweit wie möglich werden unnötige Leerzeichen dabei eliminiert. Für die einzelnen Quelltextzeilen gelten dabei folgende Vereinbarungen:

- Bei der Eingabe einer Zeile wird hinter der Zeilennummer ein Minuszeichen eingegeben.
- Jede Quelltextzeile enthält höchstens einen Assemblerbefehl.
- Vor einem Assemblerbefehl darf in derselben Zeile höchstens ein Label stehen.
- Label beginnen direkt hinter dem Minuszeichen.

- Vor jedem Assemblerbefehl steht mindestens ein Leerzeichen.
- Label und Assemblerbefehl werden durch mindestens ein Leerzeichen getrennt.
- Ein Label darf nicht allein in einer Zeile stehen.
- Ein Kommentar wird durch ein Semikolon vom Rest der Zeile getrennt.
- Reine Kommentarzeilen müssen als erstes Zeichen hinter dem Minuszeichen ein Semikolon aufweisen.
- Pseudo-Ops (.ba, .eq, etc.) können direkt hinter dem Minuszeichen beginnen.

Zur komfortablen Eingabe und Bearbeitung des Quelltextes stehen folgende Befehle zur Verfügung:

```
/a 100,10 - Automatische Zeilennumerierung
```

Hier mit der Startzeile 100 und der Schrittweite 11. Abgeschaltet wird die Funktion, indem man RETURN direkt hinter dem Minuszeichen eingibt. Man kann auch Änderungen in einer anderen Zeile vornehmen, wobei deren Nummer als neue Startnummer genommen wird. Weiterhin ist es möglich, eine neue Startnummer am Zeilenanfang einzugeben, ohne die Auto-Funktion verlassen zu müssen.

```
/o - Renew eines Quelltextes
```

Nach einem NEW-Befehl oder Reset kann der Quelltext wieder zurückgeholt werden. Nach einem Prozessorabsturz funktioniert dies jedoch im allgemeinen nicht mehr.

```
/d;/d 50;/d-50;/d 50-;/d 50-200 - Löschen von Zeilenbereichen
```

Diese Funktion bietet ähnlich wie der LIST-Befehl die Möglichkeit, Bereiche des Quelltextes zu löschen, nur daß hier gelöscht statt gelistet wird. Mit »/d« wird also der komplette Quelltext gelöscht. Man sollte auch einzelne Zeilen mit diesem Befehl löschen, da man sonst neben der betreffenden Zeilennummer ja immer das Minuszeichen mit eingeben müßte.

```
/e;/e 100;/e-100;/e 100-;/e 100-200 - Formatiertes Listen
```

Im Gegensatz zum normalen LIST-Befehl bietet diese Funktion die Möglichkeit, den Quelltext formatiert auszugeben. Dabei werden die Assemblerbefehle, Label und Kommentare gemäß den Tabulatoren übersichtlich untereinander gelistet. Man kann die Ausgabe durch die SHIFT-Tasten steuern, wobei nicht nur eine Verlangsamung, sondern auch ein völliges Anhalten möglich ist.

```
/t 0,13; /t 1,24; /t 2,0; /t 3,10 - Tabulatoren setzen
```

T0: Tabulator für Assemblerbefehle
T1: Tabulator für Kommentare
T2: Tabulator für Leerzeichen am Zeilenanfang
T3: Tabulator für Symboltabelle

```
/x - Verlassen des Assemblers durch Reset
```

```
/p 1,100,200 - Setzen eines Arbeitsbereiches (Page)
```

Hier Bereich 1 von Zeile 100 bis 200 (beide einschließlich). Bis zu 30 Arbeitsbereiche sind erlaubt. Mit dieser Funktion kann man den Quelltext in übersichtliche Teilstücke aufteilen. Die folgenden Funktionen beziehen sich immer auf einen solchen Arbeitsbereich.

```
/1,2 - Formatiertes Listen einer oder mehrerer Pages
```

Die Pages 1 und 2 werden formatiert nach Tabulatoren ausgegeben. Der Vorteil gegenüber dem »/e«-Befehl besteht darin, daß man sich keine Zeilennummern, sondern nur noch die Ziffern der einzelnen Pages merken muß.

```
/n 1,100,10 - Neunumerierung einer Page
```

Hier wird Page 1 mit der Startzeile 100 und der Schrittweite 10 neu durchnumeriert.

```
/f 1,"string" - Suchen einer Zeichenkette in einer Page
```

Im String ist das Fragezeichen als Joker erlaubt. Es ersetzt beliebige Zeichen. Es ist zu beachten, daß die unnötigen Blanks nach der Eingabe einer Zeile entfernt werden. Daher kann der zu suchende String von dem eingegebenen abweichen.

```
/r 1,"string1","string2" - Ersetzen von Zeichenketten
```

Überall in der Page 1 wird die Zeichenkette aus »string2« durch die aus »string1« ersetzt. »string2« darf nicht leer sein. Auch hier ist in »string2« das Fragezeichen als Joker erlaubt. Da »string1« leer sein darf, können mit dieser Funktion Zeichenketten auch gelöscht werden.

```
/u 16384 - Setzen des Quelltextstartes
```

Die Voreinstellung beträgt 11540 ($2D14). Der Quelltext kann maximal bis $9FFF reichen. Falls das erzeugte Maschinenprogramm im Bereich von $2D14 bis $9FFF liegen soll, muß

man den Quelltextstart verlegen, wie hier z.B. nach $4000 (16384). Damit kann man sein Maschinenprogramm in dem Bereich von $2D14 bis $3FFF unterbringen.

```
/b - Anzeige der aktuellen Speicherkonfiguration
```

Es werden angezeigt:

- der normale Quelltextstart (11540)
- der aktuelle Quelltextstart
- das Quelltextende
- die Anzahl der noch freien Bytes für den Quelltext

```
/z 100 - Setzen des Quelltextstartes
```

Der Start des Quelltextes wird auf die Zeilennummer 100 gesetzt. Diese Funktion ist z.B. dann nützlich, wenn man ganze Programmteile umstellen möchte.

```
/w 100-200,10 - Verschieben eines Quelltextbereiches
```

Die Zeilen 100 – 200 (beide einschließlich) werden hinter die Zeilennummer 10 verschoben. Ab Zeile 10 wird in Zehnerschritten neu durchnumeriert. Die Page 1 wird neu gesetzt auf den neuen Blockanfang bis zum Quelltextende. Man muß darauf achten, daß Befehle auf bestimmte Zeilennummern (.go, usw.) nicht geändert werden.

```
/l"name"; /s"name"; /v"name" - Abkürzungen der Floppybefehle
```

Die Befehle LOAD, SAVE und VERIFY können wie oben abgekürzt werden, eine Angabe der Gerätenummer ist nicht erforderlich. Mit dem »/s«-Befehl kann man den Quelltext auch abschnittsweise abspeichern, indem man vor dem Namen die entsprechenden Zeilennummern angibt, z.B. /s100–200 »name«.

```
/m"name" - Anhängen von Quelltexten
```

Hier wird der bekannte MERGE-Befehl ausgeführt. Ein Programm wird an das Ende des bestehenden Quelltextes angehängt. Besonders wertvoll wird diese Funktion in Verbindung mit Makros.

```
/g 8 - Setzen der Geräteadresse
```

```
/i - Anzeige des Directorys
```

Das Inhaltsverzeichnis der Diskette wird ohne Verlust des Quelltextes eingelesen und angezeigt. Wie beim »/e«-Befehl kann die Ausgabe mit den SHIFT-Tasten gesteuert werden.

```
/k - Auslesen und Anzeige des Floppyfehlerkanals
```

```
/  - Übermittlung von Befehlen an die Floppy
```

Dies können die üblichen Befehle, z.B. zum Formatieren einer Diskette, sein. Bei Existenz eines Floppyspeeders können aber auch dessen Spezialbefehle gesendet werden.

```
/ch 6 - Setzen der Hintergrundfarbe (hier: Dunkelblau)
```

```
/cr 14 - Setzen der Rahmenfarbe (hier: Hellblau)
```

```
/! - Ausgabe der Symboltabelle in unsortierter Form
```

```
/!! - Ausgabe der Symboltabelle in sortierter Form
```

```
- Modifizierter PRINT-Befehl
```

Der normale PRINT-Befehl kann auf Grund der Tokenbildung nicht alle Labels verarbeiten. Hierfür kann der »–«-Befehl verwendet werden. Ebenso können die Funktionen (...) und (...) außerhalb des Quelltextes benutzt werden. Die normalen Basic-Funktionen können jedoch nur über den PRINT-Befehl erreicht werden.

1.4.1.2 Hypra-Ass-Variable (Label)

Der Wert einer Hypra-Ass-Variablen kann zwischen 0 und $FFFF liegen. Variablennamen können beliebig lang sein, wobei das erste Zeichen ein Buchstabe sein muß. Weitere Zeichen können Buchstaben, Ziffern oder das Hochkomma sein.

Im Zusammenhang mit der Verwendung von Makros (siehe nächster Abschnitt) muß zwischen globalen und lokalen Variablen unterschieden werden. Jede Variable erhält beim Anlegen eine sogenannte Ordnungszahl, die angibt, im wievielten Makroaufruf das Anlegen stattfand. Wenn man sich in keinem Makro befindet, ist die Ordnungszahl Null.

Variablen unterschiedlicher Ordnungszahlen sind trotz gleichen Namens nicht gleich. Daher sind alle Variablen gleicher Ordnungszahl lokal, sie gelten ja immer nur in dem Bereich, der durch die Ordnungszahl definiert ist.

Die Konstruktion mittels Ordnungszahlen dient dazu, Fehler durch doppelte Benutzung von Labeln bei mehrmaligem Aufruf von Makros zu verhindern, indem Makros bei jedem Aufruf sozusagen einen komplett neuen Satz von Labeln erhalten.

Aus einem Makro »herausgesehen« sind alle Variablen mit anderer Ordnungszahl als im Makro selbst »unsichtbar«. Um aber bequem Makros in Makros aufrufen und Label benutzen zu können, die in mehreren Makros verwendet werden sollen, gibt es die globalen Variablen. Diese sind im Gegensatz zu den lokalen unabhängig von der Ordnungszahl überall definiert.

Makronamen sind per Definition global.

Alle Hypra-Ass-Variablen sind redefinierbar, d.h., sie können durch eine Wertzuweisung jederzeit geändert werden. Eine doppelte Benutzung von Labeln vor Assemblerbefehlen (z.B. Sprungmarken) ist jedoch nicht gestattet, da dies zu einem falschen Ergebnis bei der Assemblierung führen würde.

1.4.1.3 Die Makros von Hypra-Ass

Diesen im letzten Abschnitt schon erwähnten Begriff wollen wir nun klären.

Makros sind meist kürzere Befehlsfolgen, die im Quelltext häufiger vorkommen und deshalb unter einem Makro zusammengefaßt werden. Diese Konstruktion bietet den Vorteil, daß man gleich aufgebaute Befehlsfolgen nicht jedesmal neu »per Hand« in den Quelltext einfügen, sondern das Makro nur noch aufrufen muß. Daher gehört zu jedem Makro ein Name, mit dem es aufgerufen werden kann. An jedes Makro können beliebig viele Parameter übergeben werden, deren aktueller Wert dann bei der Assemblierung in das Makro eingesetzt wird. Makros dürfen bei Hypra-Ass an jeder beliebigen Quelltextstelle definiert werden. Alle Makronamen sind global, alle Parameter und makrointernen Label sind lokal. Damit können verschiedene Makros Label und Parameter gleichen Namens verwenden.

Um den Aufbau eines Makros zu erläutern, hier ein Beispiel, das immer wieder in der Praxis benötigt wird: eine 8x8-Bit-Multiplikation:

```
  0 -.MA MLT(M1,M2)
110 -    LDA #00
120 -    LDX #08
130 -L1  ASL
140 -    ROL M2
150 -    BCC L2
160 -    CLC
170 -    ADC M1
180 -    BCC L2
```

```
190 -     INC M2
200 -L2   DEX
210 -     BNE L1
220 -.RT
```

Dieses Makro multipliziert die beiden 8-Bit-Werte M1 und M2 miteinander und liefert ein 16-Bit-Ergebnis. Das Lowbyte hiervon steht im Akku, das Highbyte in M2.

Zunächst zum Aufbau des Makros: Der Pseudobefehl .MA zeigt ein Makro an und wird von dem Makronamen »MULT« gefolgt. Danach erkennt man die Parameterliste in runden Klammern, die hier zwei durch ein Komma getrennte Werte enthält, M1 und M2. In die Parameter werden die beim Aufruf des Makros aktuellen Werte eingesetzt.

Hinter der Definitionszeile folgt der eigentliche Makroinhalt, d.h. das, was eigentlich assembliert werden soll. Ein Makro kann fast beliebig aufgebaut sein: Es können Sprünge und bedingte Verzweigungen (wie oben) ausgeführt werden, genausogut ist es aber auch möglich, weitere Makros aufzurufen. Auch Selbstaufrufe von Makros sind erlaubt. In der letzten Makrozeile muß der Pseudo-Befehl .RT stehen, der das Ende eines Makros anzeigt. Vor den Anweisungen .MA und .RT dürfen in derselben Zeile keine Labels stehen.

Der Makroaufruf wird durch den Pseudobefehl ..., gefolgt vom Makronamen und der Parameterliste, durchgeführt, z.B.

```
...MULT(3+Faktor1,4*Faktor2-6)
```

Man erkennt, daß beim Aufruf beliebige Ausdrücke (siehe auch Abschnitt »Rechnen im Quelltext«) zugelassen sind. Wichtig ist jedoch, daß in der Makrodefinitionszeile selbst nur Label zugelassen sind. So ist im obigen Beispiel z.B.

```
.MA MULT(Faktor1,Faktor2)
```

erlaubt. Verboten hingegen ist

```
.MA MULT(Faktor1*Faktor2-5,Faktor2-Faktor1*2)
```

Nun zur Funktion des Programms: Zunächst muß man sich klarmachen, wie eine Multiplikation überhaupt definiert ist. Sie gibt ja nur an, wie oft man eine bestimmte Zahl zu sich selbst addieren muß, z.B. 3*2 bedeutet:

```
3*2 = 2+2+2
```

Genauso geht man nun binär vor. Man definiert eine Zahl als »Zähler« dafür, wie oft die andere addiert werden muß. Angenommen, das 7. Bit des Faktors M1 ist gesetzt. Dies bedeutet, daß man den Faktor M2 128mal addieren muß. Allgemein gesprochen muß man also so vorgehen, daß man folgende Teilprodukte addiert:

- M1* 128, falls Bit 7 von M2 =1, sonst 0
- M1* 64, falls Bit 6 von M2 =1, sonst 0
- M1* 32, falls Bit 5 von M2 =1, sonst 0
- M1* 16, falls Bit 4 von M2 =1, sonst 0
- M1* 8, falls Bit 3 von M2 =1, sonst 0
- M1* 4, falls Bit 2 von M2 =1, sonst 0
- M1* 2, falls Bit 1 von M2 =1, sonst 0
- M1* 1, falls Bit 0 von M2 =1, sonst 0

Die Produkte M1*2 bis M1*128 erhält man durch einmaliges bis siebenmaliges Linksverschieben des Bytes M1.

In einer Schleife (X-Register als Zähler) wird in jedem Schleifendurchlauf das höchstwertige Bit von M2 mit Hilfe des ROL-Befehls in das Carry-Flag geschoben, wo man es sofort abfragen kann. Ist das Bit 0, wird nichts addiert. Der Akku dient als Summe. Normalerweise müßte man nach dem ersten Durchlauf den Akku * 128 nehmen, wenn das 7. Bit von M2 gesetzt ist. Dies erreicht man durch 7maliges Linksverschieben. Durch einen Trick kann man sich diese Arbeit jedoch sparen: Wenn man in jedem Schleifendurchlauf vor dem ROL-Befehl den Akku um ein Bit nach links verschiebt, wird das hinausgeschobene Bit in das freigewordene Bit 0 von M2 eingeblendet. Im nächsten Durchgang wird genauso verfahren. In jedem Durchgang wird dieses Verfahren wiederholt. Man kann also auch sagen, daß nach jedem Durchgang der hinzuaddierte Wert M1 um ein Bit niederwertiger wird, da z.B. im dritten Durchgang der hinzuaddierte Wert nur noch fünfmal (8 – 3) nach links verschoben wird. Nach acht Durchgängen ergibt sich also genau das Ergebnis, das wir erreichen wollten. Diese Routine wird später im Kapitel »Grafikprogrammierung« zur Berechnung von Kreisen und Ellipsen benutzt.

1.4.1.4 Rechnen im Quelltext

Hypra-Ass erlaubt die vier Grundrechenarten plus Potenzierung, die logischen Operationen NOT, AND und OR, die Vergleiche »kleiner« und »größer« sowie den Einsatz der Funktionen (...) und (...), die das Low- bzw. Highbyte eines 16-Bit-Argumentes liefern. Die logischen Operationen und Vergleiche werden wie folgt abgekürzt:

- !n! = NOT
- !a! = AND
- !o! = OR
- !=! = gleich
- !<! = kleiner als
- !>! = größer als

Das Ergebnis eines Vergleichs ist –1, falls wahr, 0, falls nicht wahr, z.B.

(1!=!2) = 0, (1!=!1) = –1

Die NOT-Funktion liefert die gleichen Ergebnisse wie in Basic, z.B.

!n!1 = –2

Das Argument in den Low-/Highbyte-Funktionen muß im Bereich von 0 bis 65535 liegen.

Außer der Verwendung von Dezimalzahlen sind noch Hexadezimalzahlen, die durch ein vorangestelltes »$«-Zeichen gekennzeichnet werden, sowie Binärzahlen erlaubt, die man an dem »%«-Zeichen erkennt. Dabei ist es egal, wieviele Stellen die einzelnen Zahlen aufweisen. Weiterhin kann man überall dort, wo Bytewerte erwartet werden, Strings der Länge 1 verwenden, z.B. bei der unmittelbaren Adressierung. Erlaubt sind somit z.B.

```
123, $23, $8, $4DA3, %1101, %11, %1000100001110101, LDA #"a"
```

1.4.1.5 Die Pseudobefehle

Bei den sogenannten Pseudobefehlen handelt es sich um assemblerspezifische Anweisungen. Zwei, nämlich die Makrobefehle, haben Sie oben schon kennengelernt. Sehr ärgerlich ist die Tatsache, daß praktisch jeder Assemblerautor seine eigenen Pseudobefehle einführt, so daß man Quelltexte nicht von einem auf den anderen Assembler übertragen kann, ohne alle Pseudobefehle anzupassen. Der Sinn der Pseudobefehle liegt darin, Operationen ausführen zu können, die immer wieder benötigt werden, jedoch durch keinen Maschinenbefehl erreicht werden können, sowie zur Steuerung der Assemblierung. Hypra-Ass stellt Ihnen vergleichsweise viele Pseudobefehle zur Verfügung, mit denen Sie z.B. sogar die IF-THEN-Struktur aus dem Basic übernehmen können!

`.ba $C000` – Setzen der Startadresse der Assemblierung

`.st` – Beendet die Assemblierung

`.eq Load = $FFD5` – Weist dem Label »Load« den Wert $FFD5 zu, ohne die Ordnungszahl zu ändern.

`.gl Save = $FFD8` – Weist dem Label »Save« den Wert $FFD8 zu und erklärt das Label als global.

`.by 0,"a",$FA,%1111` – Fügt einzelne Bytewerte in das Programm ein. Erlaubt sind neben Zahlen auch Strings der Länge 1. Getrennt werden die einzelnen Werte durch Kommata.

`.wo Save+1` – Fügt Adressen im Format Low-/Highbyte in den Maschinencode ein. In unserem Beispiel würden also die Bytes $D9 und $FF eingefügt.

`.tx "text"` – Hiermit können Texte in den Quelltext aufgenommen werden. Bei der Assemblierung werden die ASCII-Codes der einzelnen Zeichen in den Maschinencode eingefügt.

`.ap "name"` – Mit Hilfe dieses Befehls ist es möglich, am Ende des Pass 2 automatisch weitere Quelltexte nachzuladen, wobei der Programmzähler aus der vorangegangenen Assemblierung erhalten bleibt.

`.co var1,var2` – Diese Anweisung bewirkt zwei Dinge: Zum einen werden die Labels nach dem Befehl an einen nachzuladenden Quelltext übergeben. Zweitens bleiben alle Quelltextzeilen bis zur ».co«-Anweisung erhalten (Common-Bereich). Steht z.B. ein Makro vor dieser Zeile, wird auch das Makro mitübergeben. Folgendes ist dabei zu beachten:

- Es sollten keine Makroaufrufe im Common-Bereich stehen, es sei denn innerhalb eines Makros.
- Die ».ba«-Anweisung sollte außerhalb des Common-Bereiches liegen, damit nach dem Nachladen nicht wieder mit der gleichen Startadresse assembliert wird.
- Wertzuweisungen an Labels sollten ebenfalls außerhalb des Common-Bereiches liegen, um Platz für den nachzuladenden Quelltext zu gewinnen.

`.ob "name,p,w"` – Hiermit kann das erzeugte Maschinenprogramm direkt auf Diskette geschrieben werden. Dies ist z.B. immer dann erforderlich, wenn es in dem Bereich liegt, den auch der Hypra-Ass benutzt. Würde direkt in den Speicher assembliert, würde sich der Hypra-Ass selbst überschreiben.

`.en` – Schließt erzeugtes Maschinenfile (CLOSE 14).

`.on` – Entspricht dem IF-THEN von Basic. Hinter ».on« folgt ein Ausdruck, ein Komma und ein zweiter Ausdruck. Ist der erste Ausdruck wahr, wird zu der Zeilennummer gesprungen, die der zweite Ausdruck angibt, z.B.

```
1000-.on faktor!=!12,2000
```

– Wenn Faktor gleich 12 ist, wird die Assemblierung in Zeile 2000 fortgeführt.

`.go 1000` – Ergibt unbedingten Sprung, hier zur Zeile 1000.

`.if` – Entspricht dem IF-THEN-ELSE von Basic. Hinter ».if« folgt ein Ausdruck. Ist dieser wahr, wird die Assemblierung hinter der ».if«-Zeile fortgesetzt, bis

`.el` – gefunden wird. Daraufhin wird

`.ei` – gesucht und dahinter die Assemblierung fortgesetzt. Ist der Ausdruck hinter ».if« unwahr, erfolgt die Assemblierung von ».el« bis ».ei«, ».el« kann auch fehlen, dann wird direkt hinter ».ei« fortgefahren. Unser Beispiel von oben könnte nun so lauten:

```
1000-.if faktor!=!12
1010-         lda #00
1020-.el
1030-         lda #02
1040-.ei
```

– Wenn »faktor« gleich 12 ist, erhält man LDA #00, sonst wird LDA #02 erzeugt. Vor den Pseudobefehlen ».if«, ».el« und ».ei« dürfen keine Labels in derselben Zeile stehen.

`.li 1,4,0` – sendet ein formatiertes Listing des Quelltextes unter der logischen Filenummer 1 an das Gerät mit der Geräteadresse 4 und der Sekundäradresse 0 (Drucker). Die Parameter entsprechen denen des OPEN-Befehls. Der ».li«-Befehl muß der erste Befehl im Quelltext sein, wenn alle Zeilen gelistet werden sollen. Die Zeilen bis einschließlich ».li« werden nicht ausgegeben. Die gelisteten Zeilen haben folgendes Format:

```
9000 AD 01 02: 1000-Label LDA $0201 ;
Kommentar
```

– Die Steuerung der Formatierung erfolgt mit dem Editorbefehl »/t«. Bei Zeilen, die Pseudobefehle enthalten, werden keine Adressen und Opcodes ausgegeben. Das Listing enthält die Kopfzeile »Hypra-Ass-Assemblerlisting«.

`.sy 1,4,0` – sendet am Ende des Pass 2 die sortierte Symboltabelle. Die Formatierung wird durch den Tabulator »/t3« gesteuert. Eine Zeile der Symboltabelle sieht wie folgt aus:

```
SAVE = $FFD8
```

– Die Symboltabelle enthält die Kopfzeile »Symbols in alphabetical order«.

`.dp t0,t1,t2,t3` – Setzt die Tabulatoren vom Quelltext aus. Für »t0« bis »t3« gilt das unter dem Editorbefehl »/t« Gesagte.

1.4.1.6 Die Assemblierung

Nach dem Start mit RUN wird das Maschinenprogramm im Speicher abgelegt, sofern es nicht mit dem .ob-Befehl zur Floppy geschickt wurde. Am Ende des zweiten Passes wird die Meldung »End of assembly«, gefolgt von der Assemblierungsdauer in Minuten, Sekunden und Zehntelsekunden ausgegeben. Dahinter folgt der Speicherbereich des erzeugten Maschinenprogramms in der Zeile »base = $XXXX last byte at $YYYY«.

In den seltensten Fällen wird man jedoch im ersten Versuch ein korrektes Maschinenprogramm erzeugen können. Hypra-Ass gibt in diesen Fällen 14 verschiedene Fehlermeldungen aus:

- **can't number term** – Ein Ausdruck kann von Hypra-Ass nicht berechnet werden. Möglicher Grund ist die falsche Abkürzung eines Operators.
- **end of line expected** – Bei der Abarbeitung einer Zeile wurde statt des Zeilenendes etwas anderes gefunden.
- **no mnemonic** – Ein Mnemonic kann nicht identifiziert werden. Bitte beachten Sie, daß Hypra-Ass keine illegalen Opcodes verarbeiten kann!
- **unknown pseudo** – Ein Pseudobefehl wurde falsch abgekürzt.
- **illegal register** – Ein Assemblerbefehl existiert in der gewählten Adressierungsart nicht mit dem gewählten Register.
- **wrong address** – Ein Assemblerbefehl existiert nicht in der gewählten Adressierungsart.
- **illegal label** – Das erste Zeichen eines Labels war kein Buchstabe.
- **unknown label** – In Pass 2 wurde ein unbekannter Labelname entdeckt.
- **branch too far** – Eine relative Verzweigung führt über eine zu große Distanz.
- **label declared twice** – Ein Labelname wurde zweimal benutzt.
- **too many labels** – Label und Quelltext passen zusammen nicht mehr in den Speicher.
- **no makro to close** – Die Anzahl der ».ma«-Anweisungen stimmt nicht mit der Anzahl der ».rt«-Anweisungen überein.
- **parameter** – Im Makroaufruf stimmt die Parameterliste nicht mit der Parameterliste der Definition überein.
- **return** – Es liegt keine Rücksprungadresse auf dem Stack, als eine ».rt«-Anweisung ausgeführt werden sollte.

1.4.1.7 Nützliche Makros für den Hypra-Ass

Wer mit dem Umfang der Programmiersprache Basic nicht zufrieden ist, kann sich Mengen an Erweiterungen kaufen, die teilweise über 100 neue Befehle zur Verfügung stellen. Als Maschinenprogrammierer, die wir auch Grund zur Unzufriedenheit hätten, da der Prozessor noch weniger Befehle aufweist als das Standard-Basic, können wir von so etwas natürlich nur träumen.

Mit dem Hypra-Ass können wir jedoch Makros entwickeln, die bestimmte nützliche Befehle simulieren. Wenn wir dann einen Quelltext entwickeln, können wir auf diese »Befehlserweiterung« zurückgreifen. Auf der Diskette zum Buch finden Sie unter dem Namen »Makros« einen Quelltext, der 19 neue »Befehle« bereitstellt. Hierbei handelt es sich um häufig »vermißte Standardbefehle«, wie z.B. Stackbefehle für die Indexregister.

TXY – Der Inhalt des X-Registers wird ins Y-Register übertragen.

TYX – Der Inhalt des Y-Registers wird ins X-Register übertragen.

PHX – Das X-Register wird auf dem Stack abgelegt.

PHY – Das Y-Register wird auf dem Stack abgelegt.

PLX – Das X-Register wird vom Stack geholt.

PLY – Das Y-Register wird vom Stack geholt.

Die vier folgenden Makros definieren einen Userstack, der an beliebige Stelle gelegt werden kann. Dazu muß im Hauptprogramm eine globale Variable mit dem Namen »USER« in der Zeropage angelegt werden. Anschließend muß in die Adresse, die die Variable repräsentiert, die Startadresse des Stacks geschrieben werden, z.B.:

```
10 -.GL USER = $FA
11 -      LDA #00
12 -      STA USER
13 -      LDA #$C0
14 -      STA USER+1
```

Hier wurde ein Userstack angelegt, der bei Adresse $C000 beginnt. Der Stackpointer steht in den Zeropage-Adressen $FA und $FB. Der Sinn eines Userstacks besteht darin, daß man neben dem »normalen« Stack einen weiteren Bereich zur Verfügung hat, in dem man Daten speichern kann, ohne auf die komfortable Verwaltung mit dem Stackpointer verzichten zu müssen.

`SHA` – Der Akkumulator wird auf dem Userstack abgelegt.

`PUSHAY` – Der Akkumulator und das Y-Register werden auf dem Userstack abgelegt.

`PULLA` – Der Akkumulator wird vom Userstack geholt.

`PULLAY` – Der Akkumulator und das Y-Register werden vom Userstack geholt.

`ADW (adresse)` – 16-Bit-Addition. Der Inhalt einer beliebigen Adresse wird zum Akkumulator (Lowbyte) und zum Y-Register (Highbyte) addiert. Das Ergebnis steht wieder im Akku (Lowbyte) und Y-Register (High-Byte).

`ADMW (adr1,adr2,summe)` – 16-Bit-Addition. Der Inhalt von »adr1« und »adr1+1« wird zum Inhalt von »adr2« und »adr2+1« addiert und das Ergebnis in der Adresse »summe« und »summe+1« abgelegt.

`SBCW (adresse)` – 16-Bit-Subtraktion. Der Inhalt von »adresse« und »adresse+1« wird vom Inhalt des Akkumulators (Lowbyte) und des Y-Registers (Highbyte) abgezogen. Das Ergebnis steht im Akku (Lowbyte) und im Y-Register (Highbyte).

`SBCMW (adr1,adr2,diff)` – 16-Bit-Subtraktion. Vom Inhalt »adr1« und »adr1+1« wird der Inhalt von »adr2« und »adr2+1« abgezogen. Das Ergebnis steht in »diff« und »diff+1«.

`INCW (adresse)` – Der Inhalt von »adresse« und »adresse+1« wird inkrementiert (16-Bit-Inkrementierung).

`DECW (adresse)` – Der Inhalt von »adresse« und »adresse+1« wird dekrementiert (16-Bit-Dekrementierung).

`LDAY (adresse)` – Der Akku wird mit dem Inhalt von »adresse« und das Y-Register mit dem Inhalt von »adresse+1« geladen.

`STAY (adresse)` – Der Inhalt des Akkus wird nach »adresse« und der des Y-Registers nach »adresse+1« geschrieben.

`LDAYI (wert)` – Akku und Y-Register werden mit »wert« unmittelbar geladen. Das Lowbyte steht dabei im Akku, das Highbyte im Y-Register.

1.4.2 Der Reassembler zum Hypra-Ass

Der von Martin Wehner entwickelte Reassembler ist das Gegenstück zum Hypra-Ass und genügt ebenso professionellen Ansprüchen. Mit ihm kann man aus einem Maschinenprogramm einen Quelltext erzeugen, der mit dem Hypra-Ass editiert, verändert und wieder assembliert werden kann. Er belegt den Speicherbereich von 8714 bis 11530 und wird automatisch mit dem Hypra-Ass eingeladen. Da der Quelltextstart erst bei 11540 beginnt, können

Hypra-Ass und Reassembler gleichzeitig zusammenarbeiten. Man kann also die reassemblierten Quelltexte sofort weiterverarbeiten und wieder assemblieren, ohne einmal nachzuladen. Diesen Komfort bieten nur sehr wenige Entwicklungssysteme, zumal der SMON auch noch gleichzeitig verwendet werden kann.

Neben dem eigentlichen Reassembler stehen noch einige Basic-Befehle zur Verfügung, mit denen man z.B. Einsprünge im Quelltext durch Label markieren kann. Es läßt sich auch vorherbestimmen, ob der Reassembler selbständig nach Tabellen suchen soll oder nicht. Weiterhin läßt sich der Aufbau des Quelltextes in einigen Punkten mitbestimmen. Alle dazu notwendigen Informationen werden dem Reassembler in einem kleinen Informationsprogramm mitgeteilt. Dafür stehen die folgenden drei Basic-Befehle zur Verfügung:

`-P adresse:` Mit diesem Befehl lassen sich Einsprungpunkte im Quelltext durch ein Label markieren. So sind Adressen, die mit SYS angesprungen werden, im Quelltext leichter auffindbar.

`-T adr1,adr2:` Hiermit können dem Reassembler die Lagen von Tabellen mitgeteilt werden. Während »adr1« auf das erste Byte zeigt, gibt »adr2« das letzte der Tabelle an. Im Quelltext erscheinen Tabellen als Folge von ».by«-Werten mit dazugehörigen ASCII-Codes, z.B.

```
100-label .by $41,$42,$43  ; "labc"
```

`-E byte:` Der E-Befehl steht am Ende des Informationsprogrammes und startet den Reassembler.

Der Aufbau des Quelltextes läßt sich geringfügig beeinflussen, indem man hinter dem E-Befehl eine Zahl zwischen 0 und 255 angibt. Hierbei handelt es sich um ein sogenanntes Informationsbyte. Die einzelnen Bits haben folgende Bedeutung:

Bit 0 gesetzt: Alle Zeropage-Adressen werden durch ein Label von nur drei Buchstaben (normal fünf) gekennzeichnet.

Bit 1 gesetzt: Nach den Befehlen RTS, RTI, BRK und JMP wird eine Kommentarzeile eingefügt, um den Quelltext übersichtlicher zu gestalten.

Bit 2 gesetzt: Bei allen Befehlen mit unmittelbarer Adressierung wird der Operand zusätzlich im ASCII-Format ausgegeben, vorausgesetzt, er liegt zwischen 32 und 96 oder zwischen 160 und 224, z.B. 100– LDA #65 ;»a«

Liegt er außerhalb dieser Bereiche, wird ein Punkt ausgegeben.

Bit 3 gesetzt: Zwischen je zwei Tabellenzeilen wird eine Kommentarzeile ausgegeben.

Bit 4 gesetzt: Der ASCII-Ausdruck bei Tabellen wird unterdrückt.

Bit 5 gesetzt: Ist dieses Bit gesetzt, werden externe Label sowie Tabellenlabel speziell gekennzeichnet. Tabellenlabeln wird ein »T« vorangestellt (statt z.B. LC000 nun TLC000), externen Labeln, also denen, die außerhalb des reassemblierten Bereichs liegen, ein »E« (ELC000).

Bit 6 gesetzt: Der Reassembler sucht selbständig nach Tabellen. Es wird kein Quelltext, sondern ein Basic-Informationsprogramm generiert, das die Start- und Endadressen aller gefundenen Tabellen enthält. Sie können dieses Programm normal listen und ändern.

Bit 7 gesetzt: Es werden die Speicherinhalte reassembliert, die sich unter dem ROM im RAM befinden. Somit kann man bis auf den Bereich von $D000-$DFFF auf das RAM des gesamten Adreßraums zugreifen.

Das Informationsprogramm wird durch die Befehle

```
SYS 8714,anfadr,endadr+1:RUN
```

gestartet. Dabei stellen »anfadr« die Anfangsadresse des zu reassemblierenden Programms und »endadr« dessen Endadresse dar. An einem Beispiel wollen wir uns das Vorgehen verdeutlichen. Wir wollen den Reassembler selbst reassemblieren. Zunächst geben wir das Basic-Informationsprogramm ein:

```
100 ←P$220A;kennzeichnet die Adresse 8714 durch ein Label
110 ←T$236B,$2657;definiert eine Tabelle
120 ←E15;startet den Reassembler und setzt Bits 0 bis 3
SYS8714,8714,11530:RUN im Direktmodus eingeben.
```

In weniger als 8 Sekunden wird ein etwa 17 Kbyte langer Quelltext generiert, der mit LIST oder – wenn der Hypra-Ass aktiviert wurde – dem »/e«-Befehl gelistet und mit RUN assembliert werden kann. Wie der Hypra-Ass, kann auch der Reassembler wahlweise Hex- oder Dezimalzahlen verarbeiten. Folgende Dinge müssen beachtet werden:

a) Der Reassembler arbeitet mit dem Hypra-Ass ausgezeichnet zusammen. Seine Lage wurde so gewählt, daß Hypra-Ass, Reassembler und Quelltext nebeneinander existieren können. Er kann jedoch auch ohne Aktivierung des Hypra-Ass benutzt werden. Je nachdem, ob der Hypra-Ass gestartet wurde, muß nach der Zeilennummer ein Minuszeichen eingegeben werden, z.B. 100– E15 bei aktiviertem Hypra-Ass, sonst aber 100 E15.

b) Aus programmtechnischen Gründen kann es vorkommen, daß im »Pass1« ein Maschinenprogramm anders reassembliert wird als in »Pass2«. Dadurch können in »Pass2« Sprungadressen auftauchen, die in »Pass1« nicht gefunden wurden. In diesem Fall wird die Adresse nicht durch ein Label, sondern durch eine Hex-Zahl dargestellt. An die Quelltextzeile werden drei Fragezeichen angehängt.

c) Bestimmte 3-Byte-Befehle, die bei der Assemblierung als 2-Byte-Befehle interpretiert werden (z.B. BIT $A900 = .BY $2C; LDA #$00), werden nicht reassembliert, sondern als 3-Byte mit vorangestelltem .BY-Pseudo-Op in den Quelltext eingefügt. Der reassemblierte Befehl wird aber als Kommentar an die entsprechende Zeile angefügt.

d) Es ist möglich, ein Programm zu reassemblieren, als ob es in einem anderen Bereich läge Dazu ist an den SYS-Befehl die tatsächliche Startadresse anzufügen. Um z.B. ein Programm, das im RAM von $D000 bis $D200 liegt, reassemblieren zu können, kann man es mit dem SMON nach z.B. $C000 verschieben. Dann wird der Reassembler mit dem Befehl

SYS $C000,$C200,$D000 gestartet. Der Quelltext sieht nun so aus, als ob das Programm von $D000 bis $D200 liegen würde.

e) Der erzeugte Quelltext kann statt in den Speicher auch direkt auf Diskette geschrieben werden. Dies ist besonders dann nützlich, wenn man gerade einen anderen Quelltext mit dem Hypra-Ass bearbeitet. Dazu muß man vor dem SYS-Befehl ein Programmfile öffnen:

```
OPEN 1,8,1,»NAME,P,W«:CMD1
```

Ganz wichtig ist es, daß nun nach dem SYS-Befehl das Basic-Informationsprogramm auf keinen Fall mehr mit dem RUN-Befehl gestartet werden darf, da durch diesen Befehl das File wieder geschlossen wird. Man muß vielmehr den GOTO-Befehl einsetzen, wobei hinter ihm die erste Zeilennummer des Informationsprogrammes stehen muß, im obigen Beispiel also GOTO 100. Nach jedem Speichern sollte man durch Drücken der RUN/STOP-RESTORE-Taste den alten Zustand wieder herstellen.

Genau wie der Hypra-Ass enthält auch der Reassembler einige Fehlermeldungen, die eine unkorrekte Bedienung anzeigen:

- **syntax error:** Ein Basic-Befehl wurde falsch eingegeben oder eine HEX-Zahl besteht aus weniger als 4 Ziffern.
- **out of memory:** Es steht zu wenig Speicherplatz für den Quelltext zur Verfügung oder es kommen mehr als 2700 Label vor.
- **illegal quantity:** Vor einer Hex-Zahl fehlt das Zeichen »$«, oder das Tabellenende liegt vor dem Tabellenanfang, oder die Tabellen überschneiden sich, oder die angegebene Adresse liegt nicht im Maschinenprogramm.
- **type mismatch:** In einer Hex-Zahl stehen falsche Hex-Ziffern. Die Adresse, die als Einsprung markiert wurde, darf nicht als Tabellenanfang oder -ende angesprochen werden. Im Informationsprogramm darf keine Adresse doppelt vorkommen.

1.4.3 Der SMON-Maschinensprachemonitor

Der von Dietrich Weineck entwickelte Maschinensprachemonitor SMON ist auf der Diskette zum Buch in vier Versionen enthalten, die sich durch ihre Lage bzw. Funktionen unterscheiden:

Name	Speicherlage	Ladebefehl	Startbefehl
SMON PLUS	$C000-$CFFF	LOAD"SMONPC000",8,1	SYS 49152
SMON PLUS	$3000-$3FFF	LOAD"SMONP3000",8,1	SYS 12288
SMON FLOPPY	$C000-$CFFF	LOAD"SMONFC000",8,1	SYS 49152
SMON ILLEGAL	$C000-$CFFF	LOAD"SMONIC000",8,1	SYS 49152

Die »Normalversion« SMON Plus ist doppelt vorhanden. In der Regel wird man den Monitor im Bereich von $C000-$CFFF betreiben. Wenn man aber ein Maschinenprogramm in diesen Bereich assemblieren und gleichzeitig den SMON im Speicher haben möchte, kann man auf die Version ab $3000 zurückgreifen. Sie kann gleichzeitig mit dem Hypra-Ass und dem Reassembler verwendet werden. Die einzige notwendige Anpassung besteht im Hochsetzen des Quelltextstartes nach $4000 durch den Befehl »/u 16384«.

Die beiden anderen SMON-Versionen bieten noch Spezialfunktionen, auf die wir am Ende dieses Abschnitts eingehen werden. Alles gilt zunächst für alle drei Versionen.

Wie es sich für einen Monitor der Spitzenklasse gehört, stellt der SMON alle die von anderen Programmen bekannten Standardfunktionen zur Verfügung. Dazu gehören z.B. die Anzeige des Speicherinhaltes in Hex-Bytes oder als Assemblercodes (Disassembler) mit Änderungsmöglichkeiten, Befehle zum Verschieben mit oder ohne Umrechnung der Adressen oder zum Laden, Speichern und Starten von Maschinenprogrammen. Nicht mehr zum Standard kann man einen kleinen Assembler sowie den integrierten Einzelschrittsimulator zählen, mit dem man ein Maschinenprogramm Schritt für Schritt abarbeiten und kontrollieren kann.

Der Monitor benötigt für alle Eingaben hexadezimale Zahlen ohne $. Bei der Eingabe von Adressen bedeutet ANF die tatsächliche Startadresse, während END die erste Adresse hinter dem gewählten Bereich angibt. Nachdem Sie den SMON geladen haben, geben Sie bitte ein NEW ein, um die Basic-Zeiger wieder korrekt zu setzen. Wenn Sie den Monitor gleichzeitig mit dem Hypra-Ass betreiben wollen, laden Sie zunächst den SMON, geben den NEW-Befehl ein und laden anschließend den Hypra-Ass.

1.4.3.1 Die Befehle des SMON

```
A ANF - Assemblierung z.B. A 6000
```

Nach Eingabe von RETURN erscheint auf dem Bildschirm die gewählte Adresse mit einem blinkenden Cursor. Alle Befehle werden so eingegeben, wie sie der Disassembler anzeigt. Die Eingabe einer Zeile wird wiederum mit RETURN abgeschlossen. Bei einer fehlerhaften Eingabe (z.B. STX 6000,Y existiert nicht) wird an den Anfang der Zeile zurückgesprungen. Sonst wird der Befehl disassembliert und nach Ausgabe der Hex-Bytes gelistet. Zur Korrektur vorhergehender Zeilen gehen Sie mit dem Cursor zur Anfangsposition hinter der Adresse zurück, schreiben den Befehl neu und gehen nach RETURN mit dem Cursor wieder in die letzte Zeile. Eine Spezialität des SMON besteht darin, daß man wie beim Hypra-Ass mit Labeln arbeiten kann, eine für Monitore höchst ungewöhnliche Eigenschaft. Ein Label besteht aus dem Buchstaben »M« für Marke und einer zweistelligen Hex-Zahl von 01 bis 30, z.B. JMP M05. Wenn Sie die Zieladresse für den Sprung erreicht haben, kennzeichnen Sie diese mit eben dieser Marke, z.B. M05 LDA $02.

Der Assembler nimmt auch einzelne Bytes an, indem Sie diese mit einem vorangestellten Punkt kennzeichnen. In diesem Modus werden die Eingaben natürlich nicht disassembliert.

Nach Beendigung des Assemblierens geben Sie »F« ein. Dann werden alle Eingaben noch einmal aufgelistet. Sie können sie bei Bedarf wie beim Disassembler angegeben korrigieren. Natürlich kann man den SMON-Assembler nicht mit dem Hypra-Ass vergleichen, für kleine, speicherorientierte Arbeiten ist er aber sehr hilfreich.

```
D ANF,END - Disassemblierung z.B. D C000,C201
```

Der Bereich von $C000 bis einschließlich $C200 wird disassembliert, d.h., die Speicherinhalte werden als Assemblercodes interpretiert und als solche angezeigt. Falls keine Endadresse angegeben wird, erscheint zunächst nur eine Zeile in folgendem Format:

ADR	HEX-BYTES	BEFEHL
C000	A5 FA	LDA $FA

Mit der Leertaste wird der jeweils nächste Befehl angezeigt. Wenn man eine fortlaufende Ausgabe wünscht, muß man RETURN drücken. Die Ausgabe wird nun so lange fortgesetzt, bis eine weitere Taste gedrückt wird. Mit RUN/STOP gelangt man jederzeit in den Eingabemodus zurück.

Das Komma, das vor der Adresse auf dem Bildschirm erscheint, ist ein »hidden command« (verstecktes Kommando). Es braucht nicht eingegeben werden, da es automatisch am Zeilenanfang ausgegeben wird. Man kann nun sehr einfach Änderungen im Speicher vornehmen. Dafür fährt man mit dem Cursor auf den zu ändernden Befehl, d.h. unmittelbar auf den Assemblercode, nicht auf die Hex-Bytes. Dann überschreiben Sie den Befehl mit dem neuen und drücken RETURN. Der SMON erkennt nun das Komma als Befehl und führt diesen im Speicher aus. Problematisch wird die Sache nur, falls der neue Befehl eine andere Länge als der vorhergehende aufweist. Falls er kürzer ist, müssen die unnützen Bytes mit NOPs aufgefüllt werden. Wenn er aber länger ist, muß man vorher den nachfolgenden Bereich nach hinten verschieben, um genug Platz für die neuen Bytes zu schaffen. Es ist immer ratsam, den geänderten Bereich zur Kontrolle anschließend nochmals zu disassemblieren.

```
G ANF - Starten eines Maschinenprogramms z.B. G A000
```

Das Maschinenprogramm muß mit einem Break-Befehl abgeschlossen werden, damit ein Rücksprung in den SMON erfolgen kann. Wird nach »G« keine Adresse angegeben, benutzt SMON die, welche nach dem letzten BRK erreicht wurde und bei der Registerausgabe im Programmzähler auftaucht. Mit dem »R«-Befehl (siehe unten) kann man die Register vorher auf die gewünschten Werte setzen.

```
M ANF,END - Memory-Dump z.B. M 2000,3B01
```

Der Speicherinhalt von $2000 bis einschließlich $3B00 wird in Zeilen von jeweils 8 Byte in zweistelligen Hex-Ziffern und den dazugehörenden ASCII-Zeichen ausgegeben. Wie bei der Disassemblierung kann man die Ausgabe mit der Leer- und RETURN-Taste steuern, wobei

die Endadresse wiederum fehlen darf. Die ausgegebenen Bytes können durch Überschreiben geändert werden, nicht aber die ASCII-Zeichen. Dafür ist der Doppelpunkt am Zeilenanfang verantwortlich, ein weiterer »hidden command«. Wenn die Änderung nicht durchgeführt werden kann, weil z.B. versucht wurde, ins ROM zu schreiben, wird ein Fragezeichen als Fehlermeldung ausgegeben.

```
R - Registeranzeige
```

Dieser Befehl zeigt die Inhalte der 6510-Register an. Dabei gelten folgende Abkürzungen:

AC: Akkumulator
XR: X-Register
YR: Y-Register
PC: Programmzähler
SR: Statusregister
SP: Stackpointer

Die einzelnen Flags des Statusregisters werden mit »1« für gesetzt und »0« für nicht gesetzt gekennzeichnet. Durch Überschreiben kann man die Registerinhalte ändern. Die Flags können jedoch nicht einzeln, sondern nur durch ein Überschreiben des Statusregisters geändert werden.

```
I/O-Set - Setzen der Gerätenummer z.B. I 01
```

»I 01« legt die Device-Nummer für LOAD und SAVE auf 1 (Kassette). Jedes Laden und Speichern erfolgt auf das so angegebene Gerät, bei den einzelnen Lade- und Speicheroperationen braucht die Gerätenummer daher nicht mehr angegeben zu werden. Die voreingestellte Nummer ist 8.

```
L"name"(,ANF) - LOAD z.B. L"SMON PC000",C000
```

L»Name« lädt ein Programm vom angegebenen Gerät an die Originaladresse. Die Basic-Zeiger bleiben beim Ladevorgang unverändert. Wenn man nach dem Programmnamen noch eine Adresse angibt, wird es nicht an die Originaladresse, sondern an die angegebene geladen. Damit hat man z.B. die Möglichkeit, Autostart-Programme, die in den Stackbereich geladen werden, an einer anderen Adresse untersuchen zu können.

```
S"name",ANF,END - SAVE z.B. S"SMON",C000,D000
```

Der Speicherbereich von $C000 bis einschließlich $CFFF wird auf dem angegebenen Gerät unter dem Namen SMON abgespeichert. Hierbei ist es besonders wichtig zu beachten, daß das Byte der Adresse $D000 nicht mehr berücksichtigt wird.

```
Printer-Set - Setzen der Druckergerätenummer z.B. P 02
```

Die Gerätenummer des Druckers wird auf 2 gesetzt. Voreingestellt ist hier die Nummer 4 (für seriellen Bus). Bei allen Ausgabebefehlen wie D, M, etc. können Sie neben der Ausgabe auf den Bildschirm gleichzeitig einen Ausdruck erhalten, wenn Sie das Kommando geshiftet eingeben.

```
# Dezimalzahl - Umrechnung Dezimal nach Hex z.B. #10,#61456
```

Es ist möglich, Dezimalzahlen bis 65535 in Hexadezimalzahlen umzurechnen.

```
$ Hexzahl - Umrechnung Hex. nach Dez. und Binär z.B. $1A,$FA23
```

Die Eingabe ist nur zwei- oder vierstellig erlaubt. Falls die Zahl kleiner als 256 (=$0100) ist, wird zusätzlich auch der Binärwert ausgegeben. In unserem Beispiel würde daher der erste Wert auch binär ausgegeben.

```
% Binärzahl - Umrechnung Binär nach Dez. und Hex. z.B. %10011101
```

Bei diesem Befehl muß eine genau achtstellige Binärzahl eingegeben werden. Falls einmal versehentlich mehr Stellen eingegeben werden, werden nur die ersten acht zur Berechnung herangezogen.

```
? Zahl1+-Zahl2 - Addition,Subtraktion z.B. 12A0+2C12,67FE-210D
```

Es ist möglich, zwei genau vierstellige Hexadezimalzahlen zu addieren bzw. zu subtrahieren.

```
= ANF END - Vergleichen von Speicherinhalten z.B. = C000 C200
```

Der Speicherinhalt ab $C000 wird mit dem ab $C200 byteweise verglichen. Tritt der erste unterschiedliche Wert auf, wird die zugehörige Adresse angezeigt und abgebrochen.

```
W ANF END ANFneu - Speicherber.verschieben z.B. W C000 D000 4000
```

Der Speicherbereich von $C000 bis einschließlich $CFFF wird nach $4000 verschoben. Da keine absoluten Adressen (z.B. LDA $C200,X) umgerechnet werden, ist das verschobene Programm im allgemeinen nicht mehr lauffähig.

```
O ANF END HEX-Wert - Speicherbereich füllen z.B. O 2000 3001 A0
```

Der Bereich von $2000 bis einschließlich $3000 wird mit dem Wert $A0 gefüllt. Hiermit ist es neben dem Löschen des Speichers mit dem Wert 0 auch möglich festzustellen, welche Speicherzellen von einem Programm benutzt werden. Dazu füllt man den »verdächtigen« Bereich mit einem ungewöhnlichen Wert wie z.B. $EF, läßt das Programm ablaufen und kann dann mit dem »M«-Befehl feststellen, welche Speicherzellen verändert wurden.

```
V ANFalt ENDalt ANFneu ANF END - Umrechnung von Adressen z.B.
V C000 D000 4000 4100 4300
```

Dieser kompliziert aussehende Befehl ist im Prinzip einfach zu verstehen. Angenommen, Sie haben ein Programm im Bereich von $C000 bis einschließlich $CFFF liegen und möchten es nach $4000 verschieben. Während das eigentliche Programm nur von $C100 bis $C300 reicht, sind von $C000 bis $C0FF und von $C301 bis $CFFF Tabellen mit Texten usw. angelegt. Die ersten drei Parameter kennen Sie schon vom »W«-Befehl. Sie legen den zu verschiebenden Bereich sowie die neue Startadresse fest. Die beiden weiteren Parameter geben an, welcher Bereich auf die neue Adresse umgerechnet werden soll. Hierfür kommt natürlich nur das eigentliche Programm, das ja nach der Verschiebung von $4100 bis $4300 zu finden ist, in Frage, da die Tabellen natürlich nicht geändert werden dürfen. So wird dann z.B. der Befehl LDA $C200,X in den Befehl LDA $4200,X abgeändert.

```
C ANFalt ENDalt ANFneu ANF END - Konvertierung
```

Dieser Befehl führt die Befehle »W« und »V« gleichzeitig aus, d.h., ein Speicherbereich wird zunächst verschoben und dann angepaßt. Die Parameter entsprechen denen des »V«-Befehls.

```
B ANF END - Umwandlung in DATA-Zeilen z.B. B C000 D000
```

Der Bereich von $C000 bis einschließlich $CFFF wird in DATA-Werte umgerechnet und ab Zeilennummer 32000 im Basic-Speicher abgelegt. Die Zeilennummern vorher können z.B. mit einem Basic-Ladeprogramm belegt werden. Um ein Maschinenprogramm »gebrauchsfertig« z.B. in einer Zeitschrift zu präsentieren, hat es sich bewährt, dieses in Form eines Ladeprogramms zu tun, wo die einzelnen Codes in Datas abgelegt sind, z.B.

```
10 FORI=49152TO49156:READA:POKEI,A:NEXT:SYS49152
20 DATA 169,65,76,210,255
```

Dieses Mini-Maschinenprogramm gibt ein »A« auf dem Bildschirm aus. Viele Top-Zeitschriften, wie z.B. die 64'er, sind heute jedoch dazu übergegangen, sogenannte Checker zu verwenden.

```
K ANF END - Ausgabe von ASCII-Zeichen z.B. K A09E A19E
```

In vielen Fällen kann man den Disassembler nicht anwenden, nämlich dann, wenn es gilt, den Inhalt einer Tabelle mit Texten darzustellen. Mit diesem Befehl werden die einzelnen Speicherinhalte als ASCII-Code interpretiert. Wie bei den anderen Befehlen auch, können Änderungen durch einfaches Überschreiben vorgenommen werden. Unser Beispiel listet übrigens die Basic-Befehle des C64 aus dem ROM.

```
F HEX-WErt(e),ANF END - Hex-Werte suchen z.B. F A9 00,C000 D000
```

Mit diesem Befehl ist es möglich, einen Speicherbereich nach einzelnen oder mehreren Werten zu durchsuchen. Die Bereichsangabe darf auch weggelassen werden, es wird dann der gesamte Speicher durchsucht. (Dies gilt auch für alle folgenden »F«-Befehle). Zwischen dem Zeichen »F« und dem ersten Hex-Zeichen sowie zwischen je zwei Hex-Bytes müssen unbedingt Leerzeichen stehen. In unserem Beispiel werden alle Speicherstellen gelistet, die die Kombination A9 00 (=LDA #00) enthalten.

```
FA Adresse,ANF END - sucht absolute Adressen
```

Es werden alle Befehle ausgegeben, die eine bestimmte Adresse als absoluten Operanden aufweisen. Die Adresse braucht nicht vollständig angegeben zu werden, es kann das Jokerzeichen »*« verwendet werden.

1. Beispiel: FA FFD5,C000 D000 – Im Bereich von $C000 bis einschließlich $CFFF werden alle Befehle gesucht, die $FFD5 im Operanden haben, z.B. JSR $FFD5, LDA $FFD5,X usw.

2. Beispiel: FA D***,C000 D000 – Im selben Bereich werden alle Befehle gesucht, die auf den Bereich von $D000 bis $DFFF zugreifen. Durch eine Modifikation in FA D0**,C000 D000 würde die Adresse auf den Bereich von $D000 bis $D0FF eingeschränkt.

```
FR Adresse,ANF END - sucht relative Sprünge
```

Im Gegensatz zu den absoluten Adressen werden die relativen Sprünge durch einen Offset angegeben, so daß der »FA«-Befehl wirkungslos bliebe. Der »FR«-Befehl funktioniert genauso wie dieser und läßt ebenfalls Joker zu.

1. Beispiel: FR C320,C000 D000 – Im Bereich von $C000 bis einschließlich $CFFF werden alle relativen Sprünge gesucht, die als Zieladresse $C320 aufweisen. Da diese Befehle maximal 128 Byte vom Sprungziel entfernt sein können, ist der zu durchsuchende Bereich hier viel zu groß gewählt worden. Den SMON stört dies jedoch nicht.

2. Beispiel: FR C32*,C000 D000 – Derselbe Bereich wird nach Sprüngen durchsucht, die den Bereich von $C320 bis $C32F als Ziel haben.

```
FT ANF END - Suche von Tabellen z.B. FT C000 D000
```

Im angegebenen Bereich werden Tabellen gesucht. Dabei wird alles das als solche interpretiert, was sich nicht disassemblieren läßt. Man muß jedoch grundsätzlich vorsichtig sein, da es sich auch um illegale Opcodes handeln könnte.

```
FZ Adresse,ANF END - Suche von Zeropage-Adressen
```

Alle Befehle, die eine Zeropage-Adressierung aufweisen, werden gefunden, wobei auch der Joker wieder verwendet werden darf.

1. Beispiel: FZ FA,C000 D000 – Alle Befehle, die $FA adressieren, werden gefunden, z.B. LDA ($FA),Y, ASL $FA, etc.

2. Beispiel: FZ F*,C000 D000 – Alle Befehle, die den Bereich von $F0 bis $FF adressieren, werden gefunden.

3. Beispiel: FZ **,C000 D000 – Alle Befehle, die eine Zeropage-Adressierung aufweisen, werden gefunden.

```
FI Operand,ANF END - Unmittelb. Adress. z.B. FI 02,C000 D000
```

Alle Befehle, die im angegebenen Bereich eine unmittelbare Adressierung mit dem Operanden $02 vollziehen, werden gefunden, wie LDA #02, LDX #02, LDY #02.

```
X - Verlassen des SMON
```

Es wird ins Basic zurückgesprungen. Alle Basic-Pointer bleiben erhalten.

```
ST START STOP - Trace-Stop z.B. TS C56F CCA2
```

Dieser Befehl ermöglicht die Ausführung eines Maschinenprogramms mit Unterbrechung. Dabei wird es an der Adresse START gestartet und in Echtzeit so lange ausgeführt, bis die Adresse STOP erreicht wurde. Anschließend wird in die Registeranzeige gesprungen. Mit den Befehlen »G«, »TW« oder »TB« (letztere werden später erklärt) ohne Adreßangabe kann man dann im Programmablauf fortfahren, da sich der SMON die zuletzt bearbeitete Adresse merkt.

Sinnvoll ist dieser Befehl immer dann, wenn man einen unklaren oder fehlerhaften Programmteil im Einzelschrittmodus durchlaufen will, vorher aber ein gewisser Teil bereits durchlaufen werden muß, um z.B. Benutzereingaben zu holen. Der »TW«-Befehl funktioniert nur im RAM, da nur dort ein Unterbrechungspunkt gesetzt werden kann.

```
TW START - Trace Walk z.B. TW 3FAD
```

Hierbei handelt es sich um einen Befehl zur Einzelschrittsimulation. Der erste Befehl in der Adresse $3FAD wird ausgeführt, anschließend werden alle Registerinhalte sowie der nächstfolgende Befehl angezeigt. Durch den Druck einer Taste kann man nun ein Programm Schritt für Schritt abarbeiten. Eine Besonderheit liegt in der Taste »J«. Sie dient dazu, Subroutinen, die für Sie uninteressant sind, auf einen Schlag abzuarbeiten, wie z.B. die Bildschirmausgabe $FFD2. Prinzipiell ist ein Gebrauch der »J«-Taste auch mitten in Unterprogrammen erlaubt. Man muß jedoch darauf achten, daß vorher in dem Unterprogramm keine Werte auf den Stack geschoben wurden, da sonst eine falsche Rücksprungadresse entstehen würde.

Wenn ein BRK-Befehl auftaucht, wird automatisch mit einem Sprung in die Registeranzeige abgebrochen. Sie können diesen Abbruch aber auch jederzeit durch die STOP-Taste erreichen. Anschließend können Sie wie bei »TS« beschrieben fortfahren.

Im Gegensatz zu »TS« funktioniert die Einzelschrittsimulation auch im ROM, da keine Unterbrechungspunkte gesetzt werden. Da dieser Modus jedoch interruptgesteuert ist, darf in dem Programm kein »SEI«-Befehl vorkommen. In diesem Fall muß mit der STOP-Taste abgebrochen und hinter dem SEI-Befehl fortgefahren werden. Allerdings arbeitet das Programm dann nicht mehr korrekt.

Aus diesem Grund muß man leider viele Einschränkungen in Kauf nehmen. So können z.B. keine Ein-/Ausgaberoutinen simuliert werden, da die Bedienung des seriellen Busses aus Zeitgründen natürlich nur ohne Interrupt funktioniert.

```
TB Adresse Anzahl der Durchläufe - Trace Break
```

Oft ist es ermüdend, lange Schleifen im »TW«-Modus ausführen zu müssen, da nur die Wirkung nach einer bestimmten Anzahl von Durchläufen interessiert. Der »TB«-Befehl dient dazu, eine Adresse nur eine bestimmte Anzahl mal zu durchlaufen und dann in den Einzelschrittmodus zu springen, z.B. TB C200 0A. Hier wird nach dem zehnten Durchlauf der Adresse $C200 abgebrochen. Für den Start eines solchen Programms existiert ein spezieller Befehl:

```
TQ Adresse - Trace Quick z.B. TQ C000
```

Durch diesen Befehl wird ein Maschinenprogramm gestartet, welches durch den »TB«-Befehl unterbrochen werden kann.

1.4.3.2 Die speziellen Befehle des SMON Plus

Die folgenden Befehle beziehen sich ausschließlich auf die Version SMON Plus. Mit ihnen ist es unter anderem möglich, auf einfachste Weise Sprites und neue Zeichensätze zu erstellen.

```
Q Adresse - Kopieren des Zeichensatzes in das RAM z.B. Q 9000
```

Der Zeichensatz des C64 ist für uns und den SMON normalerweise unerreichbar, da man nur über ein Abschalten des Betriebssystems an ihn herankommt. Durch diesen Befehl kann man ihn in das RAM kopieren und dort modifizieren. Um den geänderten Zeichensatz zu aktivieren, muß man dem Videocontroller die Startadresse mitteilen. Zuständig ist hierfür die Adresse $D018, den genauen Aufbau dieses Registers können Sie im Kapitel »Interruptprogrammierung« nachlesen.

```
Z ANF END - Zeichendaten ausgeben z.B. Z 2000 3000
```

Der Speicherinhalt von $2000 bis einschließlich $2FFF wird als Zeichendaten interpretiert. Jeweils ein Byte pro Zeile wird in 8-Bit-Form dargestellt. Dabei ist ein »*« ein gesetztes, ein ».« ein nicht gesetztes Bit. Jeweils 8 Byte (Zeilen) stellen ein Zeichen dar. Man kann dieses nun nach seinen Wünschen abändern, indem man einfach einen Punkt bzw. einen Stern überschreibt. So könnte man z.B. ein »A« in ein »Ä« abändern, indem man in der obersten Zeile des Zeichens zwei Punkte durch Sterne ersetzt, die dann die Punkte des »Ä« darstellen.

Weiterhin ist dieser Befehl geeignet, bestimmte Steuerbits in VIC, CIA etc. anschaulich zu beeinflussen.

```
H ANF END - Spritedaten ausgeben z.B. H 2000 3000
```

Dieser Befehl entspricht dem Befehl »Z« mit dem Unterschied, daß 3 Byte pro Zeile ausgegeben werden. Dies entspricht dem Format der Spritedaten. Damit kann man den SMON als Editor für Spriteentwürfe benutzen.

```
N ANF END - Ausgabe von Bildschirmcode (32 Z) z.B. N 0400 0600
```

Diese Funktion entspricht dem Befehl »K«. Die Daten werden jedoch nicht als ASCII-, sondern als Bildschirmcode interpretiert.

```
U ANF END - Ausgabe von Bildschirmcode (40 Z) z.B. U 0400 0600
```

Hier werden die Daten nicht mehr im Format von 32 Zeichen pro Zeile ausgegeben, sondern so, wie sie tatsächlich auf dem Bildschirm erscheinen, nämlich 40 Zeichen pro Zeile. Damit ist man in der Lage, ganze Bildschirme im Speicher zu editieren. Man kann die Daten dann

einfach bei Bedarf in den Bildschirmspeicher übertragen. Dies geht sehr viel schneller als die Ausgabe über den ASCII-Code und wird in vielen professionellen Spielen genutzt.

```
E ANF END - Löschen von Speicherbereichen z.B. E C000 D000
```

Der Bereich von $C000 bis einschließlich $CFFF wird mit Null-Bytes gefüllt.

```
Y Adresse - Verschieben des Monitors z.B. Y 60
```

Mit Hilfe dieses Befehls können Sie den SMON Plus theoretisch an 16 verschiedene Plätze im Speicher verschieben. Gewertet wird nämlich immer nur das obere Nibble (hier: $6). Sie können den verschobenen Monitor dann mit G 6000 starten.

```
J - bringt Ausgabebefehl zurück
```

Der letzte Ausgabebefehl (D,H,K,M,N,U,Z) wird nochmals angezeigt. Durch Drücken der RETURN-Taste kann man ihn wieder ausführen.

1.4.3.3 Spezielle Befehle des SMON Illegal

Diese Version ist in der Lage, bis auf fünf Ausnahmen alle illegalen Opcodes zu disassemblieren. Die Ausnahmen sind die Befehle A11, ASR, ARR, AXS und LAR. Der Opcode KIL wird als CRA und der Opcode AAX als SAX angezeigt. Die illegalen Codes werden mit einem vorangestellten Stern angezeigt, um sie von den legalen Opcodes zu unterscheiden. Um Ihnen die Verwendung nicht schmackhaft zu machen, wurde darauf verzichtet, auch den Assembler zu modifizieren. Sie können daher nur illegale Codes durch legale überschreiben, nicht aber umgekehrt.

1.4.3.4 Spezielle Befehle des SMON Floppy

```
Z - Schaltet den Diskettenmonitor ein
```

Der integrierte Diskettenmonitor wird aktiviert. Die Rahmenfarbe ändert sich auf Gelb, der gewohnte Punkt am Anfang einer Zeile wird durch einen Stern ersetzt.

```
R Track Sektor - Lesen eines Diskettensektors z.B. R 12 01
```

Der angegebene Block der Diskette wird in den Computer eingelesen und im Bereich von $BF00 - $BFFF abgelegt. Track- und Sektornummer müssen als zweistellige Hex-Zahlen eingegeben werden. Die ersten acht Byte des Blocks werden als Hexzahlen ausgegeben. Statt der Leertaste wird die Ausgabe hier mit der SHIFT-Taste gesteuert. Sie können die Bytes durch Überschreiben ändern. Eine Änderung auf der Diskette tritt aber erst dann ein, wenn der Block

mit dem folgenden Befehl zurückgeschrieben wird. Falls hinter dem »R« keine weiteren Angaben folgen, wird der logisch nächste Block eingelesen.

```
W Track Sektor - Schreiben eines Diskettensektors z.B. W 12 01
```

Der Pufferinhalt von $BF00 bis $BFFF wird auf die Diskette zurückgeschrieben. Falls hier keine Angaben über Track und Sektor gemacht werden, werden die Daten des letzten »R«-Befehls benutzt.

```
M - Anzeige des im Puffer befindlichen Blocks
```

Der zuletzt eingelesene Block wird nochmals angezeigt. Wie beim »R«-Befehl, können Sie die Ausgabe mit SHIFT und STOP steuern sowie Änderungen durch Überschreiben vornehmen.

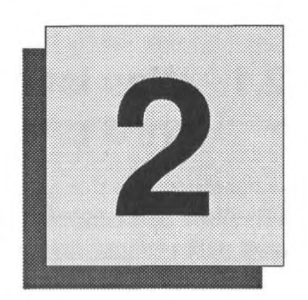
```
@- Anzeige des Floppyfehlerkanals
```

Der Fehlerkanal wird ausgelesen und ausgegeben, wenn ein Fehler vorlag. Die Meldung 00,OK, 00,00 wird unterdrückt.

```
X - Verlassen des Diskettenmonitors
```

Es wird in den SMON zurückgesprungen. Die Rahmenfarbe wird wieder auf Blau geändert und am Zeilenanfang erscheint der gewohnte Punkt. Wenn Sie nun auf den Puffer unter dem Basic-ROM zugreifen möchten, müssen Sie zunächst den Basic-Interpreter abschalten. Dies geschieht durch eine Änderung der Speicherzelle $0001 in $36 (am besten mit »M«-Kommando).

Um die Befehle des Diskettenmonitors sinnvoll einsetzen zu können, ist es empfehlenswert, sich über den genauen Aufbau einer Diskette zu informieren. Hierfür bietet sich das Buch »Die Floppy 1541« vom Markt & Technik Verlag an, in dem u.a. die Organisation des Directorys und der Files beschrieben wird.



Interrupt-programmierung von A – Z

Obwohl es inzwischen eine Menge Bücher über das Thema »Maschinensprache für den C64« gibt, ist bisher noch in keinem einzigen eine wirklich komplette Beschreibung der Interruptprogrammierung erschienen. Dies ist um so erstaunlicher, als daß diese in fast jedem professionellen Programm Verwendung findet, da sich mit ihr Effekte erzielen lassen, die mit »normaler« Programmierung nicht möglich sind.

Viele C64-Besitzer stehen der Interruptprogrammierung jedoch skeptisch gegenüber, da Programmierfehler fast immer zum totalen Systemabsturz führen. Die folgenden 50 Seiten sollen daher an Hand von einfachen Beispielen zeigen, wie man die Interrupts für sich nutzen kann. Bevor Sie jedoch mit dem Lesen beginnen, sollten Sie von der beiliegenden Diskette das Demoprogramm mit dem Befehl

```
LOAD"INTERRUPTDEMO",8
```

laden und mit »RUN« starten. Die meisten Beispiele können Sie nun einfach vom erscheinenden Menü aus anwählen und aktivieren. Dies ist der denkbar einfachste Weg. Die Demoprogramme zu den Kapiteln 2.3.1. und 2.5. müssen mit einem einfachen SYS-Befehl gestartet werden.

Eigene Interruptprogramme sollten Sie sicherheitshalber vor dem Start auf Diskette abspeichern, da immer die Gefahr eines Absturzes besteht. Ich persönlich jedenfalls wurde schon oft mit Murphys 7. Gesetz konfrontiert, bevor die ersten Interruptprogramme funktionierten.

7. Gesetz von Murphy: Die Programmentwicklung wächst so lange, bis sie die Fähigkeit des Programmieres übertrifft, der sie weiterführen muß.

Quelle: A. Bloch, Der Grund, warum alles schiefgeht, was schiefgehen kann, Goldmann 1977.

2.1 Was ist ein Interrupt und wodurch wird er ausgelöst ?

Das Wort »Interrupt« kommt aus dem Englischen und heißt »Unterbrechung«. Unterbrochen wird dabei immer das gerade ablaufende Maschinenprogramm. Ein solches läuft ständig in dem Commodore 64 ab, auch wenn Sie davon nichts bemerken: Im Eingabemodus z.B. harrt der Rechner in einer Eingabeschleife der Dinge, die da auf ihn zukommen werden, ein ablaufendes Basic-Programm wird vom Interpreter in ein Maschinenprogramm übersetzt.

Beim Interrupt handelt es sich um eine reine Hardware-Angelegenheit, in die man unmittelbar gar nicht eingreifen kann. Er kann nämlich nur durch einen Impuls an Pin 3 oder Pin 4 des 6510-Prozessors ausgelöst werden. Dabei trägt Pin 3 die Bezeichnung

```
IRQ (Interrupt Reqest)
```

Pin 4 wird

```
NMI (Non Maskable Interrupt)
```

genannt.

Der Unterschied zwischen diesen beiden Interrupt-Arten besteht darin, daß der IRQ im Gegensatz zum NMI maskierbar ist, was nichts anderes bedeutet, als daß er softwaremäßig unterdrückt werden kann. Während bei einem Impuls an Pin 4 auf jeden Fall ein NMI ausgeführt wird, wird bei einem Impuls an Pin 3 zunächst vom Prozessor geprüft, ob das sogenannte Interrupt-Flag (kurz: I-Flag) gesetzt ist. Ist das I-Flag (Bit 2 im Statusregister) gesetzt, wird die Interruptanforderung ignoriert und das vorher unterbrochene Maschinenprogramm fortgesetzt. Nur wenn das I-Flag gelöscht ist, kann ein IRQ ausgeführt werden.

Sehr naheliegend ist jetzt die Frage, wie man das Interrupt-Flag softwaremäßig beeinflussen kann. Dafür stellt der Prozessor zwei 1-Byte Befehle zur Verfügung, die zwei Taktzyklen zur Ausführung ihrer »Arbeit« benötigen:

```
SEI (Set Interrupt Mask)
```

setzt das I-Flag und verhindert damit einen IRQ, falls an Pin 3 des Prozessors ein Impuls auftritt.

```
CLI (Clear Interrupt Mask)
```

stellt das genaue Gegenteil zu SEI dar: Durch diesen Befehl wird das Interrupt-Flag gelöscht und damit der IRQ freigegeben.

So, jetzt wissen Sie, wodurch ein Interrupt ausgelöst wird und wie man ihn beeinflussen kann. Was aber passiert bei einem IRQ und NMI?

Bei einem Impuls am NMI-Pin wird der augenblicklich abgearbeitete (Maschinen-) Befehl zu Ende geführt (was bedeutet, daß »mitten« in der Ausführung eines Basic-Befehls unterbrochen werden kann). Dann wird der Inhalt des Programmzählers, d.h. die Rücksprungadresse, auf dem Stack abgelegt (erst Highbyte, dann Lowbyte). Dies ist notwendig, damit der Rechner weiß, wohin er nach der Beendigung des Interrupts springen muß. Anschließend wird der Prozessorstatus auf den Stapel geschoben, um nach dem Interrupt einen ordnungsgemäßen Fortgang des unterbrochenen Programms zu ermöglichen. Schließlich wird ein indirekter Sprung ausgeführt, dessen Zieladresse sich aus dem Inhalt der Adressen $FFFA (Lowbyte) und $FFFB (Highbyte) ergibt: JMP ($FFFA).

Bei einem Impuls am IRQ-Pin passiert im Prinzip dasselbe: Der augenblicklich abgearbeitete Maschinenbefehl wird zu Ende geführt, dann wird jedoch im Gegensatz zum NMI zunächst das Interrupt-Flag geprüft. Ist dieses gesetzt, wird einfach an der Unterbrechungsstelle mit der Abarbeitung des laufenden Maschinenprogramms fortgefahren. Ist jedoch das I-Flag gelöscht, wird ähnlich wie beim NMI verfahren: Die Rücksprungadresse und der Prozessorstatus werden auf dem Stack abgelegt, anschließend wird der indirekte Sprung JMP ($FFFE) ausgeführt.

Sowohl bei NMI als auch IRQ fällt auf, daß die indirekten Sprungadressen im ROM des Commodore 64 liegen, von uns also nicht beeinflußt werden können. Bevor wir uns ausführlich mit der IRQ bzw. NMI-Routine auseinandersetzen, wollen wir klären, wie ein Interrupt beendet werden kann. Auch dafür steht ein 1-Byte Maschinenbefehl zur Verfügung, der jedoch 6 Taktzyklen zur vollständigen Verarbeitung benötigt:

```
RTI (Return from Interrupt)
```

Er macht rückgängig, was bei einem Interruptaufruf geschah: Zunächst wird der alte Prozessorstatus vom Stack geholt und ins Statusregister geschoben. Dann wird der Wert des Programmzählers vom Stapel geholt und zurückgeschrieben (zuerst Lowbyte, dann Highbyte). Anschließend wird das unterbrochene Programm fortgesetzt, als »wäre nichts geschehen«, es hat von allem zwischen Interruptaufruf und RTI nichts bemerkt (wenn doch, hat der Interrupt-Programmierer einen Fehler gemacht). Gerade diese Spanne zwischen Interruptaufruf und RTI ist das Interessante für den Programmierer und wird in den nächsten Punkten sehr ausführlich behandelt. Alle wichtigen Grundsätze des Interrupts sind nun bekannt. Ich möchte zum Schluß dieses Punktes ausdrücklich darauf hinweisen, daß alles bisher Gesagte nicht rechnerspezifisch ist, sondern für alle Computer gilt, die einen Prozessor der Familie 65xx besitzen. Zunächst passiert also beispielsweise auf einem Apple II das gleiche wie auf einem Commodore 64 !!

2.2 Der NMI und seine »Quellen«

Bisher war immer von »Impulsen« an den Interruptpins 3 und 4 die Rede. Wodurch diese Impulse ausgelöst werden, war bisher von untergeordnetem Interesse. Um jedoch die Interrupts für eigene Zwecke nutzen zu können (nichts anderes wollen wir), müssen Sie die eigentlichen Ursachen, die »Interruptquellen« kennen. Diese Quellen sind nicht mehr allgemeingültig und bleiben (in gewissem Rahmen) der Phantasie der Konstrukteure der Computer überlassen. Beim Commodore 64 gibt es zwei Quellen, durch die ein NMI ausgelöst werden kann:

a) der I/O-Baustein CIA 2 6526 (ab Adresse $DD00)
b) die RESTORE-Taste

Bevor wir uns jedoch mit der CIA 2 beschäftigen, wollen wir einmal sehen, was genau bei einem NMI passiert. Die eigentliche NMI-Routine wird wie gesagt durch einen indirekten Sprung JMP ($FFFA) angesprungen. Die Zieladresse erhält man sehr einfach durch den Basic-Befehl

```
PRINT PEEK (65530) + 256 * PEEK (65531),
```

worauf das Ergebnis 65091 oder $FE43 auf dem Bildschirm erscheint. Hier also beginnt die eigentliche NMI-Routine, die wir uns einmal genau anschauen wollen:

```
FE43 78       SEI          IRQ wird gesperrt
FE44 6C 18 03 JMP ($0318) JMP $FE47
```

Schon wieder ein indirekter Sprung, werden Sie sagen, und dann noch ein scheinbar unsinniger, weil er auf den direkt folgenden Befehl zeigt !

In der Tat aber ist dieser Befehl der Schlüssel zur gesamten NMI-Programmierung: Im Gegensatz zu dem ersten NMI-Sprung JMP ($FFFA) wird hier die Zieladresse aus den Speicherstellen 792 (Lowbyte) und 793 (Highbyte) geholt, also aus dem RAM. Dort können wir aber hineinschreiben, den »NMI-Vektor« also beliebig abändern. Dieses werden wir uns später reichlich zu Nutzen machen, zunächst aber weiter mit der NMI-Routine.

```
FE47 48       PHA          Akku auf Stapel retten
FE48 8A       TXA
FE49 48       PHA          X-Register auf Stapel retten
FE$A 98       TYA
FE4B 48       PHA          Y-Register auf Stapel retten
```

Im Gegensatz zur Rücksprungadresse und dem Prozessorstatus werden bei der Auslösung eines NMI der Akku sowie X- und Y-Register nicht automatisch auf den Stapel gerettet. Dies muß bei einer Veränderung (die sich kaum umgehen läßt) innerhalb der Interruptroutine vom Programmierer »per Hand« durchgeführt werden.

```
FE4C A9 7F    LDA #$7F
FE4E 8D 0D DD STA $DD0D
```

Durch diesen Befehl wird die CIA 2 angewiesen, keine weiteren NMI-Anforderungen zuzulassen. Nicht betroffen davon ist die RESTORE-Taste.

```
FE51 AC 0D DD   LDY $DD0D
FE54 30 1C      BMI $FE72
```

Hier wird geprüft, was die Quelle des NMI war: War die RESTORE-Taste die Ursache, ist Bit 7 des Kontrollregisters der CIA 2 Null und es wird normal fortgefahren. War jedoch die CIA 2 die NMI-Quelle, ist dieses Bit 1 und es erfolgt ein Sprung.

```
FE56 20 02 FD   JSR $FD02     Prüft auf ROM-Modul ab $8000
FE59 D0 03      BNE $FE5E     Kein Modul, weiter mit Sprung
FE5B 6C 02 80   JMP ($8002)
```

Falls ein Modul vorhanden ist, wird ein weiterer indirekter Sprung vollzogen. Die Startadresse wird durch die Speicherstellen $8002 (Lowbyte) und $8003 (Highbyte), die sich im Modul befinden, vorgegeben. Sie können daher mit der RESTORE-Taste eigene Maschinenprogramme starten, deren Startadresse Sie in den Adressen $8002 und $8003 ablegen. Voraussetzung ist jedoch, daß Sie dem Rechner ein eingelegtes Modul »vorgaukeln«: Dafür brauchen Sie nur die Bytes 195, 194, 205, 56 und 48 ab Speicherstelle $8004 zu schreiben. (Dies bedeutet »CBM 80« und ist für den Commodore 64 das Zeichen, daß ein Modul eingelegt ist.)

Es folgt der Abschnitt, der durchlaufen wird, wenn die RESTORE-Taste die NMI-Quelle ist.

```
FE5E 20 BC F6   JSR $F6BC
```

Diesen Programmteil durchläuft auch die IRQ-Routine ! Die Funktion hängt mit der Tastaturabfrage zusammen, betreffend die Stop-Taste.

```
FE61 20 E1 FF   JSR $FFE1
```

Hierbei handelt es sich um die Stop-Routine. Auch diese wird über einen indirekten Sprung JMP ($0328) angesteuert, normalerweise zeigt dieser RAM-Vektor auf die Adresse $F6ED. Dort wird geprüft, ob die RUN/STOP Taste gedrückt wurde.

```
FE64 D0 0C      BNE $FE72
```

Wenn RUN/STOP nicht gedrückt wurde, also die RESTORE-Taste allein, wird zu der Adresse gesprungen, die schon oben angesprungen wurde, falls die Quelle des NMI die CIA 2 und nicht die RESTORE-Taste war. Zunächst jedoch folgt der Abschnitt, der bei gleichzeitigem Druck von RUN/STOP und RESTORE abgearbeitet wird.

```
FE66 20 15 FD   JSR $FD15
FE69 20 A3 FD   JSR $FDA3
FE6C 20 18 E5   JSR $E518
```

In diesem NMI-Teil werden die I/O-Bausteine, der Videocontroller, der Bildschirmeditor und die Vektoren initialisiert. Das bedeutet, daß eventuelle Zeigerveränderungen im RAM rückgängig gemacht werden und diese wieder die alten Werte enthalten.

```
FE6F 6C 02 A0  JSR ($A002)
```

Der vierte und letzte indirekte Sprung in Verbindung mit der NMI-Routine löst einen Basic-Warmstart aus. Wenn also RUN/STOP und RESTORE gedrückt wurden, ist die NMI-Routine damit beendet. Ist jedoch RESTORE allein oder die CIA 2 die NMI-Quelle, wird mit der Bearbeitung der RS232-Schnittstelle fortgefahren, deren einzelne Darstellung den Rahmen dieses Kapitels sprengen würde. Es sei nur soviel gesagt, daß diese Routine von $FE72 bis $FEBB reicht.

```
FEBC 68         PLA
FEBD A8         TAY      Y-Register vom Stapel holen
FEBE 68         PLA
FEBF AA         TAX      X-Register vom Stapel holen
FEC0 68         PLA      Akku vom Stapel holen
FEC1 40         RTI
```

Damit ist auch dieser Zweig der NMI-Routine beendet. Nachdem Akku und X- sowie Y-Register vom Stapel geholt wurden und auch Rücksprungadresse und Prozessorstatus wieder auf die alten Werte gesetzt wurden (durch den Befehl RTI, s.o.), wird der NMI beendet und die Abarbeitung des unterbrochenen Programms kann fortgesetzt werden.

Sie sehen, daß die NMI-Routine recht vielfältige Aufgaben zu erledigen hat. Für uns als Interrupt-Programmierer ist es jedoch nur wichtig zu wissen, daß wir durch das »Umlenken« des NMI-Vektors, der sich im RAM an den Speicherstellen 792 und 793 befindet, unsere eigene NMI-Routine ausführen können. Denn mit unserer Unterbrechungsroutine wollen wir ja nicht die RS232-Schnittstelle bedienen oder die RUN/STOP-Taste überprüfen, sondern für uns sinnvolle Aufgaben ausführen, nicht wahr? Seien wir also den Programmierern des Betriebssystems für diesen zuerst sinnlos erscheinenden indirekten Sprungbefehl dankbar und kommen wir zum Aufbau der für uns wichtigen NMI-Quelle, der CIA 2.

2.2.1 Die NMI-Quelle CIA 2

Der Complex Interface Adapter (CIA) ist ein universeller Baustein, der im einzelnen über folgende Merkmale verfügt:

- 16 einzeln programmierbare I/O-Leitungen
- 2 unabhängige, kaskadierbare 16-Bit-Timer
- eine 24-Stunden-Echtzeituhr mit programmierbarer Alarmzeit
- ein 8-Bit-Schieberegister für den seriellen Bus

Die CIA besitzt 16 Register, die vom Prozessor wie fortlaufende Speicherstellen behandelt werden können. Die CIA 2, die für einen Impuls am NMI-Pin verantwortlich ist, liegt im Adreßraum von $DD00 bis $DD0F. Ein NMI kann dabei durch die Echtzeituhr, die Timer und hardwaremäßig durch das serielle Schieberegister und einen Impuls am Pin 24 ausgelöst werden. Da innerhalb dieses Kapitels jedoch nur auf Software-Interrupts eingegangen werden soll (allen Lesern, die sich für die Hardwaremöglichkeiten interessieren, sei das Buch »Der

Commodore 64 und der Rest der Welt« empfohlen, in dem reichlich von der Interruptprogrammierung Gebrauch gemacht wird), interessieren wir uns ausschließlich auf den NMI durch Echtzeituhr und durch die beiden 16-Bit-Timer. Um diese Vorgänge zu verstehen, sehen wir uns am besten zuerst die CIA 2 einmal genau an, bevor wir dann zum ersten Beispiel, der 24-Stunden-Echtzeituhr kommen:

Register 0 Die Bits 0–7 entsprechen dem Zustand der Pins 0–7 des Port A.

Register 1 Die Bits 0–7 entsprechen dem Zustand der Pins 0–7 des Port B.

Register 2 Mit diesem Register kann jede der 8 Datenleitungen des Port A wahlweise auf Ein- (Bit=0) oder Ausgabe (Bit=1) gesetzt werden.

Register 3 Dieses Register entspricht in Belegung und Funktion dem Register 2, bezogen auf Port B.

Register 4 Zugriff: Lesen
Der augenblickliche Stand des Lowbytes des Timer A wird wiedergegeben.
Zugriff: Schreiben
Man kann in das Register das Lowbyte hineinschreiben, von dem auf Null heruntergezählt werden soll.

Register 5 Entspricht in Belegung und Funktion dem Register 4, jedoch auf das Highbyte bezogen.

Register 6 Entspricht dem Register 4, jedoch auf Timer B bezogen.

Register 7 Entspricht dem Register 5, jedoch auf Timer B bezogen.

Register 8 Echtzeituhr 1/10 Sekunden
Zugriff: Lesen
Bits 0–3: Zehntelsekunden der Uhr im BCD-Format.
Bits 4–7: Immer Null.
Zugriff: Schreiben, wobei Bit 7 Reg 15=0
Bits 0–3: Eingabe der Zehntelsekunden der Uhrzeit im BCD-Format
Bits 4–7: Müssen immer Null sein.
Zugriff: Schreiben, wobei Bit 7 Reg 15=1
Bits 0–3: Vorwahl der Zehntelsekunden der Alarmzeit im BCD-Format
Bits 4–7: Müssen immer Null sein.

Register 9 Echtzeituhr Sekunden
Zugriff: Lesen
Bits 0–3: Einersekunden der Uhr im BCD-Format.
Bits 4–7: Zehnersekunden der Uhr im BCD-Format.
Zugriff: Schreiben, wobei Bit 7 Reg 15=0
Bits 0–3: Eingabe der Einersekunden der Uhrzeit im BCD-Format.
Bits 4–7: Eingabe der Zehnersekunden der Uhrzeit im BCD-Format.

Zugriff: Schreiben, wobei Bit 7 Reg 15=1
Bits 0–3: Vorwahl der Einersekunden der Alarmzeit im BCD-Format.
Bits 4–7: Vorwahl der Zehnersekunden der Alarmzeit im BCD-Format.

Register 10 Echtzeituhr Minuten
Die Belegung und Funktion entspricht der des Registers 9, jedoch auf die Minuten der Uhr bezogen.

Register 11 Echtzeituhr Stunden
Zugriff: Lesen
Bits 0–3: Einerstunden der Uhr im BCD-Format.
Bit 4: Zehnerstunde der Uhr im BCD-Format.
Bits 5–6: Immer Null
Bit 7: 0=vormittags, 1=nachmittags.
Zugriff: Schreiben, wobei Bit 7 Reg 15=0
Bits 0–3: Eingabe der Einerstunden im BCD-Format.
Bit 4: Eingabe der Zehnerstunde im BCD-Format
Bits 5–6: müssen immer Null sein.
Bit 7: Eingabe der Tageshälfte, 0=vormittags, 1=nachmittags.
Zugriff: Schreiben, wobei Bit 7 Reg 15=1
Bits 0–3: Vorwahl der Einerstunde der Alarmzeit im BCD-Format.
Bit 4: Vorwahl der Zehnerstunde der Alarmzeit im BCD-Format.
Bits 5–6: müssen immer Null sein.
Bit 7: Vorwahl der Tageshälfte der Alarmzeit: 0=vormitt., 1=nachmitt.

Register 12 Ein- und Ausgabe für Daten, die über den seriellen Port hinein-/herausgeschoben werden.

Register 13 Interrupt Control Register (ICR)
Zugriff: Lesen (Interrupt-Data)
Bit 0=1: Unterlauf von Timer A
Bit 1=1: Unterlauf von Timer B.
Bit 2=1: Uhrzeit = Alarmzeit.
Bit 3=1: serielles Schieberegister bei Eingabe gefüllt bzw. bei Ausgabe leer.
Bit 4=1: Impuls am Pin 24 der CIA.
Bits 5–6: Immer Null.
Bit 7: =1, wenn mindestens ein Bit der Interrupt-Daten mit dem entsprechenden Bit der Interrupt-Maske übereinstimmt.
Zugriff: Schreiben (Interrupt-Mask)
Bits 0–4: Durch Setzten dieser Bits kann man auswählen, wodurch ein NMI ausgelöst werden soll, wobei die Bedeutung der einzelnen Bits der des Lesezugriffs entspricht
Bits 5–6: müssen immer Null sein.

Bit 7: =1, dann Freigabe der durch die Bits 0–4 festgelegten Funktion für NMI. =0, dann sperrt ein gesetztes Bit die entsprechende NMI-Möglichkeit.

Register 14 Control Register A

Bit 0: 0=Timer A Stop, 1=Timer A Start.

Bit 1: 1=Unterlauf von Timer A wird an Pin Port B 6 angezeigt.

Bit 2: 0=Unterlauf von Timer A erzeugt an Pin Port B 6 einen High-Impuls.

Bit 3: 0=Timer A zählt nur einmal auf Null und bleibt dann stehen (Shot-Mode). 1=Timer A wird nach jedem Unterlauf wieder mit dem Startwert geladen und gestartet (Continuous-Mode)

Bit 4: Timer A wird unbedingt mit einem neuen Startwert geladen.

Bit 5: 0=Timer A wird durch Systemtakt, 1= durch Pin 40 getriggert.

Bit 6: 0=serieller Port ist Eingang. 1=serieller Port ist Ausgang.

Bit 7: 0=Echtzeituhrfrequenz ist 50 Hz, 1= 60 Hz.

Register 15 Control Register B

Bits 0-4: entsprechen in Belegung und Funktionen den Bits 0–4 des Registers 14, bezogen auf Timer B und Pin Port B 7.

Bits 5–6: 00= Timer B wird durch Systemtakt, 01= durch Pin 40 getriggert. 10= Timer B zählt Unterläufe Timer A. 11= Timer B zählt Unterläufe Timer A, wenn Pin 40 = 1 ist.

Bit 7: 0=Echtzeituhrmodus Uhrzeit setzen, 1=Echtzeituhrmodus Alarmzeit setzen.

2.2.1.1 Die Echtzeituhr der CIA 2 als NMI-Auslöser

Die 24-Stunden-Echtzeituhr der CIA 2 hat eine Auflösung (Genauigkeit) von 1/10 Sekunde und stellt mit Abstand das genaueste »Zeitmessungsmittel« des Commodore 64 dar. Um so erstaunlicher ist es, daß das Betriebssystem weder von dieser, noch von der in der CIA 1 vorhandenen Uhr Gebrauch macht. Dies zeigt sich z.B. in der inneren Uhr TI$, deren Ganggenauigkeit genauso groß ist, wie die einer mit Solarzellen betriebenen Uhr bei bewölktem Himmel. Wir können daher beide Echtzeituhren des Rechners uneingeschränkt nutzen, was bei den Timern nicht der Fall ist, dazu aber später.

Die Uhr ist in den Registern 8 bis 11 organisiert:

- Register 8 die Zehntelsekunden,
- Register 9 die Sekunden,
- Register 10 die Minuten und
- Register 11 die Stunden.

Binär	**Dezimal**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Das Interessante ist, daß die weiteren Werte 1010 bis 1111 (dezimal 10–15) in dem BCD-Format nicht genutzt werden. Ist eine Dezimalzahl größer als 9, besitzt also 2 Stellen, so wird die Zehnerstelle (die erste Ziffer) getrennt behandelt, so daß z.B. die Zahlen 10–15 folgendermaßen dargestellt würden:

Binär		**Dezimal**
Zehnerstelle	Einerstelle	
0001	0000	10
0001	0001	11
0001	0010	12
0001	0011	13
0001	0100	14
0001	0101	15

Damit die Uhrzeit korrekt eingegeben werden kann, muß die folgende Reihenfolge eingehalten werden: Zuerst muß das Stundenregister beschrieben werden, da dabei die Uhr automatisch anhält. Dann können Minuten und Sekunden eingegeben werden. Zuletzt muß das 1/10-Register beschrieben werden, da damit auch die Uhr gestartet wird. Auf diese Weise ist gewährleistet, daß die Uhr auch tatsächlich zur gewünschten Zeit losläuft.

Nach dieser Beschreibung der Echtzeituhr soll jetzt (endlich) die Möglichkeit des Auslösens eines NMI erläutert werden: Dafür ist das Interrupt-Control-Register (ICR) der CIA 2 zuständig. Dieses Register erfüllt zwei Funktionen: In der »Betriebsart« INT MASK können Sie durch Hineinschreiben die Quelle für den NMI festlegen. Wie aus der Registerbelegung ersichtlich ist, muß man Bit 2 dieses Registers setzen, damit im Alarmfall (programmierte Alarmzeit = tatsächliche Uhrzeit) ein Impuls am NMI-Pin ausgelöst wird. Zusätzlich muß man das 7. Bit setzen, um den NMI durch Alarm zu erlauben.

Programmtechnisch sieht dieses so aus:

```
LDA %10000100       Bits 2 und 7 im
STA $DD0D           ICR-Register setzen
```

Die zweite Funktion des Registers 13 besteht darin, durch das Auslesen (Betriebsart INT DATA) festzustellen, ob ein gewisses Ereignis eingetreten ist. In unserem Beispiel wird also Bit 2 gesetzt, wenn die Uhrzeit mit der gewählten Alarmzeit übereinstimmt. Bit 7 wird immer dann gesetzt, wenn mindestens ein Bit aus INT DATA mit INT MASK übereinstimmt.

Da in unserem Fall in beiden Modi das 2. Bit gesetzt ist, wird auch dieses 7. Bit gesetzt. Durch Lesen des ICR kann man also feststellen, ob der NMI durch die CIA 2 ausgelöst wurde, dann nämlich, wenn das 7. Bit gesetzt ist. An dieser Stelle möchte ich jedoch darauf hinweisen, daß dieses Register durch das Auslesen gelöscht wird! Wenn man den Inhalt später also noch benötigt, muß man ihn unbedingt zwischenspeichern.

Nach soviel Theorie können wir uns nun an das erste Beispielprogramm heranwagen, was genau das bisher Besprochene realisiert: Zu einer von uns gewählten Alarmzeit wird ein NMI ausgelöst, der einen netten Farbeffekt auf dem Bildschirm erzeugt. Hier zunächst der Quelltext:

Listing »echtzeit-nmi«

```
100:   c2ba           -;echtzeit-nmi
110:   c2ba           -         .ba $c2ba
121:                  -;
122:   dd0e           -         .eq crega     =    $dd0e
123:   dd0f           -         .eq cregb     =    $dd0f
124:   dd0b           -         .eq stunde    =    $dd0b
125:   dd0a           -         .eq minute    =    $dd0a
126:   dd09           -         .eq sekunde   =    $dd09
127:   dd08           -         .eq zehntel   =    $dd08
128:   dd0d           -         .eq icreg     =    $dd0d
129:   fe56           -         .eq nmialt    =    $fe56
130:   d020           -         .eq rahmen    =    $d020
135:                  -;
136:                  -;initialisierung
137:                  -;===============
138:                  -;
140:   c2ba ad 0e dd-           lda  crega      ;trigger auf
150:   c2bd 09 80   -           ora  #%10000000 ;50 hz
160:   c2bf 8d 0e dd-           sta  crega      ;setzen
170:                  -;
180:   c2c2 ad 0f dd-           lda  cregb      ;uhrzeit
190:   c2c5 29 7f   -           and  #%01111111 ;eingeben
200:   c2c7 8d 0f dd-           sta  cregb
210:                  -;
220:   c2ca 20 3e c3-           jsr  holstring  ;uhrzeit holen
```

```
240:   c2cd a5 fa   -        lda  $fa
250:   c2cf 8d 0b dd-        sta  stunde      ;stunden,
260:   c2d2 a5 fb   -        lda  $fb
270:   c2d4 8d 0a dd-        sta  minute      ;minuten,
280:   c2d7 a5 fc   -        lda  $fc
290:   c2d9 8d 09 dd-        sta  sekunde     ;sekunden,
300:   c2dc a9 00   -        lda  #00
310:   c2de 8d 08 dd-        sta  zehntel     ;zehntel setzen
320:                -;
330:   c2e1 ad 0f dd-        lda  cregb       ;alarmzeit
340:   c2e4 09 80   -        ora  #%10000000 ;eingeben
350:   c2e6 8d 0f dd-        sta  cregb
360:                -;
370:   c2e9 20 3e c3-        jsr  holstring   ;alarmzeit holen
380:                -;
390:   c2ec a5 fa   -        lda  $fa         ;stunden,
400:   c2ee 8d 0b dd-        sta  stunde
410:   c2f1 a5 fb   -        lda  $fb
420:   c2f3 8d 0a dd-        sta  minute      ;minuten,
430:   c2f6 a5 fc   -        lda  $fc
440:   c2f8 8d 09 dd-        sta  sekunde     ;sekunden,
450:   c2fb a9 00   -        lda  #00
460:   c2fd 8d 08 dd-        sta  zehntel     ;zehntel setzen
470:                -;
480:   c300 a9 84   -        lda  #%10000100 ;interrupt (nmi)
490:   c302 8d 0d dd-        sta  icreg       ;erlauben
500:   c305 a9 10   -        lda  #(nmineu)   ;Zeiger auf
510:   c307 a0 c3   -        ldy  #(nmineu)   ;neue NMI-Routine
520:   c309 8d 18 03-        sta  $0318       ;setzen
530:   c30c 8c 19 03-        sty  $0319
540:   c30f 60      -        rts
550:                -;
555:                -;neue nmi - routine
556:                -;=================
557:                -;
560:   c310 48      -nmineu  pha              ;akku auf stapel
570:   c311 8a      -        txa
580:   c312 48      -        pha              ;x-reg auf stapel
590:   c313 98      -        tya
600:   c314 48      -        pha              ;y-reg auf stapel
610:   c315 ac 0d dd-        ldy  icreg
620:   c318 98      -        tya
630:   c319 29 04   -        and  #%00000100 ;nmi durch alarm
640:   c31b c9 04   -        cmp  #04         ;ausgeloest
650:   c31d f0 03   -        beq  farbe       ;ja, farbeffekt
660:   c31f 4c 56 fe-        jmp  nmialt      ;alte nmi-routine
670:                -;
```

```
671:                -;alarm - nmi
672:                -;===========
673:                -;
680:  c322 a0 00    -farbe     ldy  #00         ;24-bit schleife
690:  c324 a2 00    -loop1     ldx  #00
700:  c326 a9 00    -loop2     lda  #00
710:  c328 8d 20 d0-loop3      sta  rahmen      ;rahmenfarbe
720:  c32b 18       -          clc              ;erhoehen
730:  c32c 69 01    -          adc  #01
750:  c32e d0 f8    -          bne  loop3
760:  c330 e8       -          inx
770:  c331 d0 f3    -          bne  loop2
780:  c333 c8       -          iny
790:  c334 c0 03    -          cpy  #03
800:  c336 d0 ec    -          bne  loop1
810:  c338 68       -          pla              ;y-register holen
820:  c339 a8       -          tay
830:  c33a 68       -          pla              ;x-register holen
840:  c33b aa       -          tax
850:  c33c 68       -          pla              ;akku holen
860:  c33d 40       -          rti              ;nmi beenden
870:                -;
871:                -;zeitstring holen
872:                -;================
873:                -;
880:  c33e 20 fd ae-holstringjsr  $aefd       ;komma
890:  c341 20 9e ad-           jsr  $ad9e       ;string holen und
900:  c344 20 a3 b6-           jsr  $b6a3       ;pruefen
910:  c347 c9 06    -          cmp  #06         ;laenge 6 zeichen
920:  c349 d0 3f    -          bne  illqu       ;nein, abbruch
930:  c34b a0 00    -          ldy  #00
940:  c34d b1 22    -          lda  ($22),y     ;1.ziffer holen
950:  c34f 38       -          sec
960:  c350 e9 30    -          sbc  #48         ;ascii nach hex
970:  c352 c9 03    -          cmp  #03         ;mehr als eine "2"
980:  c354 b0 34    -          bcs  illqu       ;ja, abbruch
990:  c356 0a       -          asl              ;in zehnerbereich
1000: c357 0a       -          asl              ;(bits 4-6)
1010: c358 0a       -          asl              ;schieben
1020: c359 0a       -          asl
1030: c35a 85 fe    -          sta  $fe         ;merken
1040: c35c c8       -          iny
1050: c35d b1 22    -          lda  ($22),y     ;2.ziffer holen
1060: c35f 38       -          sec
1070: c360 e9 30    -          sbc  #48         ;ascii nach hex
1080: c362 c9 0a    -          cmp  #10         ;mehr als eine "9"
1090: c364 b0 24    -          bcs  illqu       ;ja, abbruch
```

```
1100:  c366 05 fe    -          ora  $fe          ;stunden komplett
1110:  c368 d0 04    -          bne  weiter
1120:  c36a a9 92    -          lda  #$92         ;24 uhr
1130:  c36c d0 0f    -          bne  ok
1140:  c36e c9 24    -weiter    cmp  #$24         ;mehr als "23"
1150:  c370 b0 18    -          bcs  illqu        ;ja, abbruch
1160:  c372 c9 13    -          cmp  #$13         ;weniger als "12"
1170:  c374 90 07    -          bcc  ok           ;ja, fertig
1180:  c376 38       -          sec
1190:  c377 f8       -          sed
1200:  c378 e9 12    -          sbc  #$12         ;nachmittag-bit
1210:  c37a d8       -          cld
1220:  c37b 09 80    -          ora  #%10000000 ;setzen
1230:  c37d 85 fa    -ok        sta  $fa          ;stunde setzen
1240:  c37f 20 8d c3-          jsr  holerest     ;minuten holen
1250:  c382 85 fb    -          sta  $fb          ;und setzen
1260:  c384 20 8d c3-          jsr  holerest     ;sekunden holen
1270:  c387 85 fc    -          sta  $fc
1280:  c389 60       -          rts
1290:  c38a 4c 48 b2-illqu     jmp  $b248        ;illegal quantity
1300:  c38d c8       -holerest  iny
1310:  c38e b1 22    -          lda  ($22),y      ;3.oder 5.ziffer
1320:  c390 38       -          sec
1330:  c391 e9 30    -          sbc  #48          ;ascii nach hex
1340:  c393 c9 06    -          cmp  #06          ;mehr als "5"
1350:  c395 b0 f3    -          bcs  illqu        ;ja, abbruch
1360:  c397 0a       -          asl               ;in zehnerbereich
1370:  c398 0a       -          asl               ;(bits 4-6)
1380:  c399 0a       -          asl               ;schieben
1390:  c39a 0a       -          asl
1400:  c39b 85 fe    -          sta  $fe          ;merken
1410:  c39d c8       -          iny
1420:  c39e b1 22    -          lda  ($22),y      ;4.oder 6.ziffer
1430:  c3a0 38       -          sec
1440:  c3a1 e9 30    -          sbc  #48          ;ascii nach hex
1450:  c3a3 c9 0a    -          cmp  #10          ;mehr als "9"
1460:  c3a5 b0 e3    -          bcs  illqu
1470:  c3a7 05 fe    -          ora  $fe          ;gesamte minuten
1480:  c3a9 60       -          rts               ;oder sek. zurueck
```

Zunächst wird der 50-Hz-Modus gewählt, da unsere Uhr sonst erheblich nachgehen würde. Dann wird von der Routine »HOLSTRING« die aktuelle Uhrzeit sowie die gewünschte Alarmzeit geholt und wie oben besprochen in die dafür zuständigen Register geschrieben: Dies geschieht zum ersten bei gelöschtem Bit 7 des Registers 15 (Uhrzeit eingeben, Zeile 190), bei der zweiten Eingabe wird dieses Bit gesetzt (Alarmzeit eingeben, Zeile 340). Zum Schluß wird der schon viel gerühmte NMI-Vektor auf unsere Routine gesetzt und der NMI durch Setzen der Bits 2 und 7 des ICR für unsere Zwecke freigegeben.

In der neuen NMI-Routine ab Zeile 560 wird zunächst das nachvollzogen, was auch die »alte« NMI-Routine machte: Der Akku und X-sowie Y-Register werden auf den Stapel gerettet. Dann wird geprüft, ob der NMI durch einen Alarm ausgelöst wurde, indem das ICR gelesen wird. Dies ist unbedingt notwendig, da theoretisch ja auch die RESTORE-Taste oder indirekt der serielle Bus den NMI ausgelöst haben könnten. Ist das 2. Bit gesetzt, können wir sicher sein, daß es »unser« NMI ist, worauf in die Routine »FARBE« gesprungen wird, die den schon angekündigten Farbeffekt erzeugt und anschließend den NMI beendet. Ist das 2. Bit des ICR jedoch gelöscht, wurde der NMI durch eine andere Quelle als durch die Echtzeituhr ausgelöst und wir springen in die alte NMI-Routine. Aufgerufen wird dieses Programm mit

```
SYS 49850,A$,F$
```

Hierbei bedeuteten:

```
A$: Aktuelle Zeit im Stringformat
F$: Alarmzeit im Stringformat
```

Die Stringformate entsprechen dem der Variablen TI$. So wird z.B. die Zeit 8h 21min 4s durch den String »082104« dargestellt.

Wenn Sie im Demoprogramm den Programmteil »Echtzeitinterrupt« anwählen, können Sie sehen, daß dieses nicht nur graue Theorie ist, sondern tatsächlich funktioniert. Die Anwendungsmöglichkeiten der Echtzeituhr in Bezug auf Interrupt sind vielfältig: Vielleicht lassen Sie sich einmal morgens statt von Ihrem Wecker vom Commodore 64 wecken, falls Sie einen Monitor mit Lautsprecher besitzen: Statt eines monotonen Klingelns könnten Sie Ihre Lieblingsmelodie erklingen lassen, der C64 besitzt ja enorme Klangreserven. Genauer als die Echtzeituhr jedenfalls wird Ihr batteriebetriebener Wecker garantiert nicht sein!

2.2.1.2 Die 16-Bit-Timer als NMI-Auslöser

Der Commodore 64 verfügt über 4 Timer, die in den beiden CIAs untergebracht sind: Jede CIA enthält deren zwei, im folgenden Timer A und Timer B genannt. Was aber ist ein Timer?

Ein Timer ist ein »Instrument«, das man mit einem 16-Bit-Wert, also von 1 bis 65535, laden kann. Dieser Wert wird bei jedem Taktimpuls um eins auf Null heruntergezählt. Diese nicht sehr interessant erscheinende Tatsache wird jedoch dadurch aufgewertet, daß bei einem Unterlauf eines Timers ein Interrupt ausgelöst werden kann. Dabei wird durch die beiden Timer der CIA 1 ein IRQ ausgelöst, während die CIA-2-Timer einen NMI bewirken. Für uns sind nur die beiden Timer der CIA 2 und der Timer B der CIA 1 frei verfügbar, da der Timer A der CIA 1 durch den Systeminterrupt, den wir noch ausführlich besprechen werden, belegt ist. Deshalb werden wir auch in diesem Beispiel die CIA 2 benutzen, um beide Timer frei programmieren zu können.

Da der C64 mit einer Frequenz von 985248.4 Hz betrieben wird, ist ein Taktzyklus nicht länger als 1/985248.4 Sekunden, das sind ca. 0,000001014 Sekunden oder 1.014 Mikrosekunden. Die längste mit einem Timer erreichbare Zeitspanne beträgt daher 65535 * 1,014 Ms = 0.0665 Sekunden oder 66,5 Millisekunden. Diese Zeitspanne ist so kurz, daß man kaum brauchbare

Anwendungen programmieren kann. Um dieses Problem zu lösen, kann man Timer A und Timer B koppeln, d.h. zu einem 32-Bit-Timer zusammenfassen. Damit lassen sich Zeiten bis zu 65535 * 65535 * 1,014 Ms = 4359 Sekunden oder 1 Stunde, 12 Minuten und 39 Sekunden erreichen. Damit kann man aber sehr wohl sinnvolle Anwendungen programmieren, wie Sie in unserem Beispiel sehen werden. Die Betriebsart der Timer A und B werden in den Registern 14 und 15 der CIA 2 festgelegt. Für Timer A ist das 3. Bit des Registers 14 zuständig: Ist es gesetzt, wird der Timer im sogenannten »Shot mode« betrieben, d.h. er zählt nur einmal vom Ausgangswert herunter und bleibt dann stehen. Ist das 3. Bit jedoch gelöscht, wird Timer A im »Continuous mode« betrieben. Das bedeutet, daß nach dem Ablauf des Timers dieser erneut mit dem Startwert geladen und wiederum heruntergezählt wird. Dies wiederholt sich, bis der Timer abgeschaltet wird. Für Timer B ist das Register 15 zuständig: Dabei wird auch hier durch das 3. Bit zwischen »Shot mode« und »Continuous mode« unterschieden. Die Bits 5 und 6 legen fest, durch welche Quelle der Timer B getriggert wird:

Bit 5	**Bit 6**	
0	0	Timer zählt Systemtakte (normalerweise der Fall)
0	1	Timer zählt steigende CNT-Flanken (Pin 40 der CIA)
1	0	Timer B zählt Unterläufe von Timer A
1	1	Timer B zählt Unterläufe von Timer A, wenn CNT=1

In den letzten beiden Betriebsarten werden also beide Timer gemeinsam als 32-Bit-Timer betrieben. Dabei zählt ein Timer erst dann herunter, wenn der andere abgelaufen ist. Die zweite Betriebsart ermöglicht es, Impulse »von außen« zu zählen, was von großer Bedeutung für externe Bausteine ist, hier jedoch nicht behandelt werden soll. Für uns bleibt daher nur der dritte Betriebsfall (Timer B zählt Unterläufe von Timer A) übrig, da nur in ihm ausschließlich durch den Systemtakt getriggert wird.

Für die Werte, mit denen die Timer versorgt werden, sind folgende Register zuständig:

Register 4:	Timer A, Lowbyte
Register 5:	Timer A, Highbyte
Register 6:	Timer B, Lowbyte
Register 7:	Timer B, Highbyte

Auch diese Register haben eine Doppelbelegung: Durch Hineinschreiben (WRITE) kann man die Werte festlegen, von denen auf Null heruntergezählt werden soll. Durch das Auslesen der Register (READ) kann man den augenblicklichen Wert des Timer erfahren, z.B. durch

```
PRINT PEEK (56580) + 256 * PEEK (56581)
```

den Augenblickswert des Timer A der CIA 2.

Gestartet wird der Timer A durch das Setzen des 0. Bits des Register 14, Timer B durch Setzen des 0. Bits des Register 15. Durch Löschen dieser Bytes werden die Timer wieder gestoppt. Um z.B. Timer A mit Timer B gekoppelt zu betreiben, muß man die Befehle

```
LDA %00000001         Timer A im Continuous-mode starten
STA $DD0E
LDA %01010001         Timer B mit Timer A gekoppelt starten
STA $DD0F
```

anwenden. Das Setzen des 4. Bits des Register 14 dient dazu, Timer A nach Ablauf des Timer B unbedingt mit dem Startwert zu laden. Dieses »Zwangsladen« ist nicht erforderlich, wenn man den Timer A in der Reihenfolge Lowbyte/Highbyte lädt (konstruktionsbedingte Gründe der CIA).

Unser zweites Demoprogramm erlaubt es, frei wählbare Teile des Bildschirms in »fast« stufenlosen Zeitabständen blinken zu lassen.

Listing: »timer-nmi«

```
110:   c3ad              -;timer-nmi
120:   c3ad              -        .ba $c3ad
130:                     -;
140:   fe56              -        .eq nmialt    =    $fe56
150:   aefd              -        .eq chkcom    =    $aefd
160:   ffff              -        .eq time      =    65535
170:   b7f7              -        .eq int       =    $b7f7
171:   dd06              -        .eq tblow     =    $dd06
172:   dd07              -        .eq tbhigh    =    $dd07
173:   dd04              -        .eq talow     =    $dd04
174:   dd05              -        .eq tahigh    =    $dd05
175:   dd0e              -        .eq crega     =    $dd0e
176:   dd0f              -        .eq cregb     =    $dd0f
177:   dd0d              -        .eq icreg     =    $dd0d
180:   ad8a              -        .eq frmnum    =    $ad8a
181:   00fc              -        .eq low       =    $fc
182:   00fd              -        .eq high      =    $fd
183:   00fe              -        .eq zahl      =    $fe
184:   009e              -        .eq flag      =    $9e
190:                     -;
191:                     -;initialisierung
192:                     -;===============
193:                     -;
200:   c3ad 20 fd ae-init          jsr  chkcom     ;komma
210:   c3b0 20 8a ad-              jsr  frmnum     ;wert holen
220:   c3b3 20 f7 b7-              jsr  int        ;nach integer
230:   c3b6 a5 14    -             lda  $14        ;lowbyte null
240:   c3b8 d0 07    -             bne  ok
250:   c3ba a5 15    -             lda  $15        ;und highbyte
```

```
260:   c3bc d0 03    -         bne  ok          ;null
270:   c3be 4c 33 c4-          jmp  illqu       ;illegaler wert
280:   c3c1 a5 15    -ok       lda  $15         ;low- und
290:   c3c3 8d 07 dd-          sta  tbhigh      ;highbyte
300:   c3c6 a5 14    -         lda  $14         ;in timer b
310:   c3c8 8d 06 dd-          sta  tblow       ;schreiben
320:   c3cb a9 ff    -         lda  #(time)     ;timer a mit
330:   c3cd a2 ff    -         ldx  #(time)     ;1/60 sekunde
340:   c3cf 8d 04 dd-          sta  talow       ;laden
350:   c3d2 8e 05 dd-          stx  tahigh
351:   c3d5 20 fd ae-          jsr  chkcom      ;komma
352:   c3d8 20 9e b7-          jsr  $b79e       ;erste zeile
353:   c3db 86 fa    -         stx  $fa         ;merken
354:   c3dd e0 19    -         cpx  #25         ;= 25
355:   c3df b0 52    -         bcs  illqu       ;ja, illegal
356:   c3e1 a9 00    -         lda  #($d800)    ;start farbram
357:   c3e3 85 fc    -         sta  low         ;low -und
358:   c3e5 a9 d8    -         lda  #($d800)    ;highbyte
359:   c3e7 85 fd    -         sta  high        ;setzen
360:   c3e9 e0 00    -         cpx  #00         ;erste zeile
361:   c3eb f0 10    -         beq  fertig      ;ja
362:   c3ed a5 fc    -loop     lda  low
363:   c3ef 18       -         clc              ;zeile
364:   c3f0 69 28    -         adc  #40
365:   c3f2 85 fc    -         sta  low         ;addieren
366:   c3f4 a5 fd    -         lda  high
367:   c3f6 69 00    -         adc  #00
368:   c3f8 85 fd    -         sta  high
369:   c3fa ca       -         dex
370:   c3fb d0 f0    -         bne  loop
371:   c3fd 20 fd ae-fertig    jsr  chkcom      ;komma
372:   c400 20 9e b7-          jsr  $b79e       ;anzahl zeilen
373:   c403 e0 00    -         cpx  #00         ;null
374:   c405 f0 2c    -         beq  illqu       ;illegaler wert
375:   c407 a9 19    -         lda  #25
376:   c409 38       -         sec
377:   c40a e5 fa    -         sbc  $fa         ;zeilendifferenz
378:   c40c 85 fa    -         sta  $fa         ;merken
381:   c40e ca       -         dex              ;mehr zeilen
382:   c40f e4 fa    -         cpx  $fa         ;als moeglich
383:   c411 b0 20    -         bcs  illqu       ;illegaler wert
384:   c413 e8       -         inx
387:   c414 86 fe    -         stx  zahl        ;speichern
388:   c416 a9 11    -         lda  #%00010001;timer a
389:   c418 8d 0e dd-          sta  crega       ;starten
390:   c41b a9 51    -         lda  #%01010001;timer b mit a
391:   c41d 8d 0f dd-          sta  cregb       ;gekoppelt starten
```

```
400:    c420 ad 0d dd-        lda  icreg     ;interreg. loesch
410:    c423 a9 82   -        lda  #%10000010;nmi durch timer
420:    c425 8d 0d dd-        sta  icreg     ;erlauben
430:    c428 a9 36   -        lda  #(nmineu) ;zeiger auf neue
440:    c42a a2 c4   -        ldx  #(nmineu) ;nmi-routine
450:    c42c 8d 18 03-        sta  $0318     ;setzen
460:    c42f 8e 19 03-        stx  $0319
470:    c432 60      -        rts
471:                 -;
472:                 -;illegal quantity error
473:                 -;======================
474:                 -;
475:    c433 4c 48 b2-illqu   jmp  $b248
476:                 -;
480:                 -;
481:                 -;neue nmi-routine
482:                 -;================
483:                 -;
490:    c436 48      -nmineu  pha            ;akku auf stapel
500:    c437 8a      -        txa
510:    c438 48      -        pha            ;x-reg. auf stapel
520:    c439 98      -        tya
530:    c43a 48      -        pha            ;y-reg. auf stapel
540:    c43b ac 0d dd-        ldy  $dd0d     ;nmi durch timer
550:    c43e 98      -        tya            ;erzeugt
560:    c43f 29 02   -        and  #%00000010
570:    c441 d0 03   -        bne  blinken   ;ja
580:    c443 4c 56 fe-        jmp  nmialt    ;zum alten nmi
590: 590:                -;
591:                 -;bildschirm blinken
592:                 -;==================
593:                 -;
600:    c446 a5 9e   -blinken lda  flag      ;blinkphase
602:    c448 f0 0a   -        beq  blink
603:    c44a a9 00   -        lda  #00
604:    c44c 85 9e   -        sta  flag      ;phase 1=
605:    c44e ad 86 02-        lda  $0286     ;verschwinden
606:    c451 4c 5b c4-        jmp  out       ;der schrift
607:    c454 a9 ff   -blink   lda  #255      ;phase 2=
608:    c456 85 9e   -        sta  flag      ;hervorholen
609:    c458 ad 21 d0-        lda  $d021     ;der schrift
610:    c45b a6 fc   -out     ldx  low
611:    c45d 86 fa   -        stx  $fa       ;gewaehlten
A12:    c45f a6 fd   -        ldx  high
613:    c461 86 fb   -        stx  $fb       ;abschnitt
614:    c463 a6 fe   -        ldx  zahl
616:    c465 a0 00   -loop1   ldy  #00       ;des farbrams
```

```
617:   c467 91 fa   -loop2   sta  ($fa),y
618:   c469 c8      -        iny               ;mit blink-
619:   c46a c0 28   -        cpy  #40
620:   c46c d0 f9   -        bne  loop2        ;phasenwert
621:   c46e a8      -        tay
622:   c46f a5 fa   -        lda  $fa          ;fuellen
623:   c471 18      -        clc
624:   c472 69 28   -        adc  #40          ;(zeilenweise)
625:   c474 85 fa   -        sta  $fa
626:   c476 a5 fb   -        lda  $fb
627:   c478 69 00   -        adc  #00
628:   c47a 85 fb   -        sta  $fb
629:   c47c 98      -        tya
630:   c47d ca      -        dex
631:   c47e d0 e5   -        bne  loop1
670:   c480 68      -        pla
680:   c481 a8      -        tay               ;y-register holen
690:   c482 68      -        pla
700:   c483 aa      -        tax               ;x-register holen
710:   c484 68      -        pla
720:   c485 40      -        rti               ;akku holen
```

In der Initialisierungsroutine werden zunächst die Timer wie oben beschrieben mit den Startwerten versorgt, wobei Timer A immer mit dem Wert 65535 (ca. 66,5 Millisekunden) geladen wird. Timer B wird mit einem frei wählbaren Wert von 1 bis 65535 geladen. Dadurch läßt sich die Blinkdauer nur in Schritten von 66,5 ms, dem Wert des Timers A, festlegen, weshalb die Blinkfrequenz auch nur »fast« frei einstellbar ist. Nachdem der Bildschirmausschnitt, der blinken soll, festgelegt wurde, werden die Timer gekoppelt als 32-Bit-Timer gestartet (Zeilen 388 bis 391) und der NMI durch Unterlauf des Timer B festgelegt (Zeile 410). Dies geschieht analog zur Echtzeituhr: Während in unserem ersten Beispiel die Bits 2 (für Echtzeit-NMI) und 7 (NMI erlauben) des ICR gesetzt wurden, werden hier die Bits 1 und 7 gesetzt. Durch Bit 1 wird nämlich Timer B zur NMI-Quelle erklärt, durch Setzen des 0. Bits hätte Timer A diese Aufgabe übernommen.

Die neue NMI-Routine gleicht der unseres ersten Beispiels, nur wird hier Bit 1 des Interrupt-Control-Registers geprüft und nicht Bit 2. Die Bildschirmblinkroutine ab Zeile 600 schreibt abwechselnd die aktuelle Cursorfarbe (Schrift erscheint) und die Hintergrundfarbe (Schrift verschwindet) in das Farb-RAM, so daß der Blinkeffekt erzeugt wird. Mit RTI schließlich wird diese NMI-Routine abgeschlossen. Der Start dieser Routine erfolgt mit

```
SYS 50093,D,E,A
```

Dabei bedeuten:

D: Blinkdauer (0–65535)
E: Erste Blinkzeile (0–24)
A: Anzahl der blinkenden Zeilen (1–24)

Damit sind wir am Ende der Beschreibung des NMI angelangt. Zusammenfassend kann man sagen, daß dieser durch die CIA 2 bewirkte Interrupt durch diverse zeitliche Vorgänge ausgelöst werden kann. Obwohl auch hier schon verschiedene Möglichkeiten zur Unterbrechung vorhanden sind, stellt der NMI von den beiden Interruptarten die eingeschränkteren Möglichkeiten zur Verfügung. Damit sind wir auch schon beim nächsten Thema, dem IRQ.

2.3 Der IRQ und seine »Quellen«

Im Gegensatz zum NMI gibt es zwei Bausteine, die einen Impuls am IRQ-Pin auslösen können:

a) **Der I/O-Baustein CIA 1 6526 (ab Adresse $DC00)**
b) **Der Videocontroller VIC 6569**

Bevor wir uns jedoch mit diesen Bausteinen beschäftigen, wollen wir uns die IRQ-Routine einmal genau ansehen. Diese wird durch den indirekten Sprung JMP ($FFFE) angesprungen, so daß wir uns die Startadresse mit

```
PRINT PEEK (65534) + 256 * PEEK (65535)
```

holen können. Die eigentliche IRQ-Routine beginnt bei $FF48 (dez. 65352), sehen wir sie uns doch einmal an:

```
FF48  48         PHA          Akku auf Stapel retten
FF49  8A         TXA
FF4A  48         PHA          X-Register auf Stapel retten
FF4B  98         TYA
FF4C  48         PHA          Y-Register auf Stapel retten
```

Wie in Kapitel 2.1 beschrieben wurde, werden auch beim IRQ vom Prozessor nur der Wert des Programmzählers und der Status auf den Stack gespeichert. Die Register und der Akku müssen daher vom Programmierer in Sicherheit gebracht werden.

```
FF4D  BA         TSX
FF4E  BD 04 01   LDA $0104,X
```

Durch diese beiden Befehle wird das zu Beginn des Interrupts gerettete Statusregister gelesen. Wozu dies geschieht, sehen wir jetzt:

```
FF51  29 19      AND #$10     Isolieren des BREAK-Flags
FF53  F0 03      BEQ $FF58    nicht gesetzt
FF55  6C 16 03   JMP ($0316)  BREAK-Routine
FF58  6C 14 03   JMP ($0314)  IRQ-Fortsetzung
```

Die IRQ-Routine stellt einen Verteiler dar, je nachdem, ob das sogenannte BREAK-Flag gesetzt ist. Dies ist immer dann der Fall, wenn in einem Maschinenprogramm vom Prozessor

der Befehl BRK (Code 00) angetroffen wird. In diesem Fall wird in die sogenannte BREAK-Routine verzweigt, deren Startadresse ebenfalls durch einen RAM-Vektor (Lowbyte: 790, Highbyte: 791) vorgegeben ist. Auf die BREAK-Routine komme ich noch ausführlich zu sprechen.

Zunächst soll jedoch die »lupenreine« IRQ-Routine unser Thema sein, die immer dann angesprungen wird, wenn das BREAK-Flag gelöscht ist. Die Startadresse können wir uns wie beim NMI aus dem RAM holen, dadurch ist es auch hier möglich, den »IRQ-Vektor« auf unsere eigenen Routinen umzuleiten. Die Startadresse der IRQ-Fortsetzung (den Anfang haben wir ja oben schon gesehen) holen wir uns durch den Basic-Befehl

```
PRINT PEEK (788) + 256 * PEEK (789)
```

aus dem RAM. Das Ergebnis lautet im Normalfall 59953, daher beginnt unsere IRQ-Routine bei $EA31:

```
EA31  20 EA FF  JSR $FFEA    Stop-Taste abfragen und Uhr erhöhen
EA34  A5 CC     LDA $CC      Cursor in Blinkphase ?
EA36  D0 29     BNE $EA61    Nicht blinkend, weitermachen
EA38  C6 CD     DEC $CD      Blinkzähler vermindern
EA3A  D0 25     BNE $EA61    Noch nicht abgelaufen, weiter
```

Der Cursor blinkt nur bei jedem 20. Systeminterrupt, d.h. ca. einmal pro Sekunde. Der folgende Abschnitt wird daher nur bei jedem 20. IRQ durchlaufen.

```
EA3C A9 14      LDA #$14     Blinkzähler mit Ausgangswert
EA3E 85 CD      STA $CD      (20) versorgen
EA40 A4 D3      LDY $D3      Cursorspalte holen
EA42 46 CF      LSR $CF      Blinkschalter ein, dann C-Bit setzen
EA44 AE 87 02   LDX $0287    Farbe unter dem Cursor
EA47 B1 D1      LDA ($D1),Y  Zeichen unter dem Cursor holen
EA49 B0 11      BCS $EA5C    Blinkschalter ein, weitermachen
EA4B E6 CF      INC $CF      Blinkschalter ein
EA4D 85 CE      STA $CE      Zeichen unter Cursor merken
EA4F 20 24 EA   JSR $EA24    Zeiger auf Farbram berechnen
```

Die zu der Cursorposition korrespondierende Adresse im Farb-RAM wird in den Speicherstellen $F3 (Lowbyte) und $F4 (Highbyte) abgelegt.

```
EA52 B1 F3      LDA ($F3),Y  Farbe holen
EA54 8D 87 02   STA $0287    speichern
EA57 AE 86 02   LDX $0286    Farbe unter dem Cursor holen
EA5A A5 CE      LDA $CE      Zeichen unter dem Cursor holen
EA5C 49 80      EOR #$80     Zeichen wird invertiert (Revers)
EA5E 20 1C EA   JSR $EAC1    Zeichen und Farbe setzen
```

In dieser Routine wird der Akkuinhalt (Zeichen unter dem Cursor) auf den Bildschirm und das X-Register (Farbe unter dem Cursor) in das Farb-RAM geschrieben.

```
EA61 A5 01      LDA $01      Recordertaste gedrückt ?
```

```
EA63 29 10      AND #$10
EA65 F0 0A      BEQ $EA71    Ja
EA67 A0 00      LDY #$00     Datasettenflag löschen
EA69 84 C0      STY $C0
EA6B A5 01      LDA $01      Datasettenmotor
EA6D 09 20      ORA #$20     ausschalten
EA6F D0 08      BNE $EA79    unbedingter Sprung
EA71 A5 C0      LDA $C0      Datasettenflag gelöscht ?
EA73 D0 06      BNE $EA7B    Nein, Motor nicht einschalten
EA75 A5 01      LDA $01      Datasettenmotor
EA77 29 1F      AND #$1F     einschalten
EA79 85 01      STA $01
```

Die letzten Befehle behandeln die Datasette und sind für uns Floppybesitzer daher uninteressant.

```
EA7B 20 87 EA   JSR $EA87    Tastaturabfrage
EA7E AD 0D DC   LDA $DC0D    ICR der CIA 1 löschen
EA81 68         PLA          Y-Register vom Stapel holen
EA82 A8         TAY
EA83 68         PLA          X-Register vom Stapel holen
EA84 AA         TAX
EA85 68         PLA          Akku vom Stapel holen
EA86 40         RTI          IRQ beenden
```

Auffällig ist, daß die IRQ-Routine im Gegensatz zum NMI Aufgaben erfüllt, die für das »Funktionieren« des Rechners unmittelbar von Bedeutung sind, wie z.B. die Tastaturabfrage, ohne die keinerlei Kommunikation mit dem Commodore 64 möglich wäre. Daraus ergibt sich sofort ein wichtiger Unterschied zum NMI: Die IRQ-Routine muß aus Funktionsgründen des Computers ständig abgearbeitet werden. Da sich das Betriebssystem aber nicht darauf verlassen kann, daß wir als Programmierer für diesen regelmäßig erforderlichen IRQ sorgen werden, mußte es die Sache selbst in die Hand nehmen. Dabei handelt es sich um den schon erwähnten Systeminterrupt.

2.3.1 Der Systeminterrupt als IRQ-Quelle

Zuerst muß natürlich die Frage aufgeworfen werden, durch welche Quelle der Systeminterrupt ausgelöst wird. Da auch das Betriebssystem nur die CIA 1 und den Videocontroller zur Verfügung hat, einen IRQ auszulösen, liegt die Lösung fast schon auf der Hand: Es wird ein Timer der CIA 1 genutzt, genauer gesagt der Timer A. Dieser wird vom Computer im »Continuous mode« betrieben, so daß in regelmäßiger Folge ein IRQ ausgelöst werden kann. Der Timer wird dabei jedesmal mit dem Wert 16421 geladen, so daß zwischen je zwei IRQs eine Zeitdifferenz von ca. 1/60 Sekunde auftritt.

Durch Setzen der entsprechenden Bits im Interrupt-Control-Register der CIA 1 (deren Registerbelegung der der CIA 2 entspricht, so daß Sie alles Gesagte an dem obigen Beispiel der

CIA 2 nachvollziehen können) sorgt der Rechner selbst dafür, daß man überhaupt mit ihm arbeiten kann, indem 60mal in der Sekunde ein IRQ ausgelöst wird.

Das für den Programmierer so Interessante am Systeminterrupt besteht darin, daß dieser IRQ »einfach da ist«, man braucht ihn also nicht künstlich, wie den NMI, zu erzeugen. Man muß nur den IRQ-Vektor verbiegen und bekommt dann den IRQ »frei Haus« geliefert. Dieses Programmierprinzip wird oft als »Reinhängen« bezeichnet, da man von dem Systeminterrupt in der Regel eigene Routinen ausführen läßt und dann die normale IRQ-Routine anspringt, um z.B. die Tastaturabfrage zu ermöglichen.

Die Idee des folgenden Beispiels besteht darin, dem Cursor das oft störende Blinken abzugewöhnen, wie es in den meisten Textverarbeitungsprogrammen realisiert ist. Die Initialisierungsroutine setzt dazu den IRQ-Vektor auf unsere neue Routine. Beachten Sie bitte, daß im Gegensatz zum NMI vor einem Eingriff in diesen Vektor durch den SEI-Befehl ein IRQ unbedingt verhindert werden muß. Es besteht nämlich sonst die Gefahr, daß gerade zu dem Zeitpunkt ein IRQ ausgelöst wird, wo das Lowbyte des Vektors schon auf unsere Routine, das Highbyte jedoch noch auf die IRQ-Routine bei $EA31 zeigt. Dies ist immer möglich, da ja jeweils nach einer 1/60 Sekunde ein neuer IRQ ausgelöst wird. Nachdem der IRQ-Vektor komplett auf unsere neue Routine zeigt, kann dieser wieder zugelassen werden, womit unsere Initialisierungsroutine schon beendet wäre.

Im Listing der alten IRQ-Routine erkennen Sie, daß der Teil von $EA34 bis $EA60 für das Cursorblinken verantwortlich ist. Die erste Handlung der Routine bestand jedoch darin, die Stop-Taste abzufragen, was wir natürlich in unsere neue Routine übernehmen müssen. Dann sehen Sie die neue Cursor-Routine, die im wesentlichen eine Kürzung der »Original«-Routine darstellt, weil der für den Blinkvorgang verantwortliche Teil entfallen ist. Sie sehen nun einen stehenden Cursor auf dem Bildschirm, was Ihren Augenarzt vor Freude in Tränen ausbrechen lassen wird. Gestartet wird die Routine mit SYS 49520. Damit ist auch schon alles Wichtige über den Systeminterrupt gesagt, so daß nur noch das Listing des Quelltextes verbleibt.

Listing: »system-irq«

```
c170           -;systeminterrupt
c170           -          .ba $c170
ea61           -          .eq irqalt   =    $ea61
               -;
               -;initialisierung
               -;==============
               -;
c170 78        -          sei          ;interrrupt verhindern
c171 a9 7d     -          lda  #(irqneu)
c173 a2 c1     -          ldx  #(irqneu)
c175 8d 14 03- 
               -          sta  $0314   ;irq-vektor auf neue
c178 8e 15 03-            stx  $0315   ;routine setzen
c17b 58        -          cli          ;interrupt erlauben
c17c 60        -          rts
```

```
                   -;
                   -;neue interruptroutine
                   -;====================
                   -;
c17d 20 ea ff-irqneu    jsr  $ffea   ;stop testen
c180 a5 cc    -         lda  $cc     ;cursor in blinkphase
c182 d0 1d    -         bne  exit    ;nein, ende
c184 a4 d3    -         ldy  $d3     ;cursorspalte holen
c186 a5 cf    -         lda  $cf     ;zeichen invertiert
c188 d0 17    -         bne  exit    ;ja, ende
c18a e6 cf    -         inc  $cf     ;cursor in blinkphase
c18c 20 24 ea-          jsr  $ea24   ;farbramadresse holen
c18f b1 d1    -         lda  ($d1),y;zeichen unter cursor
c191 85 ce    -         sta  $ce     ;abspeichern
c193 49 80    -         eor  #128    ;invertieren
c195 91 d1    -         sta  ($d1),y;und wieder schreiben
c197 b1 f3    -         lda  ($f3),y;farbe holen
c199 8d 87 02-          sta  $0287   ;abspeichern
c19c ad 86 02-          lda  $0286   ;aktuelle cursorfarbe
c19f 91 f3    -         sta  ($f3),y;in farbram schreiben
c1a1 4c 61 ea-exit      jmp  irqalt ;fortsetzen
```

2.3.2 Die »restliche« CIA 1 als IRQ-Quelle

Die CIA 1 entspricht in den für uns wesentlichen Teilen völlig ihrem Gegenstück, der CIA 2. Damit wissen Sie auch schon, wie Sie diesen Baustein für Ihre Zwecke nutzen können:

Uneingeschränkt steht Ihnen die Echtzeituhr zur Verfügung, die wie oben im Kapitel 1 beschrieben, betrieben werden kann. Problematisch wird es jedoch bei den Timern: Da Timer A vom Systeminterrupt genutzt wird, kann man ihn nicht für eigene Anwendungen benutzen. Dadurch ergibt sich unmittelbar, daß Sie die Timer A und B nicht gekoppelt betreiben können, also nur Timer B zur freien Verfügung haben. Wie wir oben gesehen haben, lassen sich mit einem 16-Bit-Timer nur Zeiten bis ca. 66,5 Millisekunden realisieren, so daß Ihre Möglichkeiten gegenüber der CIA 2 erheblich zurückbleiben. Ich halte es daher für das Beste, grundsätzlich die CIA 2 für eigene Anwendungen in Bezug auf die Timer zu benutzen.

Die zweifellos interessanteste Interrupt-Quelle des Commodore 64 stellt jedoch der Videocontroller VIC dar, den wir nun behandeln wollen.

2.3.3 Der Video-Interface-Chip (VIC) als IRQ-Quelle

Der VIC des Commodore 64 ist für alles das verantwortlich, was man sehen kann: Die wichtigste Aufgabe besteht darin, den Bildschirm auf dem Monitor oder Fernseher aufzubauen. Weiterhin ist er für den Aufbau sowie die Steuerung der Sprites, und eben alles, was mit Grafik

zusammenhängt, verantwortlich. Für diese Aufgaben braucht man ein Menge Platz: Der VIC nennt daher nicht weniger als 47 (!) Register sein eigen, deren genaue Belegung Sie im Anschluß an dieses Kapitel finden. Dabei können Sie wie bei den CIAs jedes Register wie eine Speicherstelle ansprechen, die Basic-Adresse beträgt 53248 ($D000). Analog zur CIA besitzt auch der VIC mehrere Möglichkeiten, einen Interrupt (Impuls am Pin IRQ) auszulösen. Dafür zuständig sind die Register 25 (Interrupt-Request-Register (IRR)) und 26 (Interrupt-Mask-Register (IMR)). Beide Register haben dabei die gleiche Belegung:

Bit 0=1:	IRQ durch Rasterzeilen-Interrupt
Bit 1=1:	IRQ durch Sprite-Hintergrund-Kollision
Bit 2=1:	IRQ durch Sprite-Sprite-Kollision
Bit 3=1:	IRQ durch Lightpen-Impuls
Bits 4–6:	Unbenutzt
Bit 7=1:	Mindestens eins der Bits 0–3 ist 1.

Wollen Sie nun eine bestimmte IRQ-Quelle festlegen, müssen Sie nur im IMR das entsprechende der Bits 0 bis 3 und das 7. Bit setzen. Im IRR wird vermerkt, ob dieses Ereignis auch tatsächlich eingetreten ist, z.B. wird bei einer Sprite-Sprite-Kollision das 2. Bit gesetzt. Ein IRQ wird immer dann ausgelöst, wenn ein Bit sowohl im Register 25 (IRR) als auch im Register 26 (IMR) gesetzt ist, mit anderen Worten immer dann, wenn das Ereignis im IMR als Interrupt-Quelle festgelegt wurde und dann dieses Ereignis auch tatsächlich eingetreten ist, was im IRR registriert wird.

Der grundsätzliche Umgang mit dem IRR und IMR soll anhand eines Beispiels verdeutlicht werden: Nehmen wir an, Sie wollen einen IRQ durch einen Rasterzeilen-Interrupt erlauben. Wie oben beschrieben wurde, müssen Sie dazu das entsprechende Bit (hier Bit 0) und das 7. Bit setzen, so daß Ihr Befehl lauten muß:

```
LDA %10000001
STA IMR
```

Interessant ist, daß die anderen Bits (1–3) davon unbeeinflußt bleiben. Nun wollen wir uns einmal das Gegenteil ansehen, indem Sie den Rasterzeileninterrupt als IRQ-Quelle sperren wollen. Dies ist immer dann erforderlich, wenn vorher der IRQ durch Rasterzeilen erlaubt war, wie gerade besprochen. In diesem Fall müssen Sie ebenfalls das entsprechende »Quellen«-Bit setzen (hier wieder Bit 0), jedoch muß nun das 7. Bit gelöscht werden. Durch

```
LDA %00000001
STA IMR
```

würde also der IRQ durch Rasterzeilen-Interrupt nicht mehr zugelassen.

Auch hier bleiben die nicht betroffenen Bits unangetastet. Sie sehen, daß sich das IMR gewaltig von einer RAM-Speicherzelle unterscheidet, da hier die nicht angesprochenen Bits selbstverständlich beeinflußt würden.

Der wichtigste Hinweis jedoch betrifft das Register 25 (IRR): Dies wird nach Auslösen des IRQ nämlich nicht gelöscht, so daß z.B. nach einem erfolgten IRQ durch eine Sprite-Kollision nach Verlassen der IRQ-Routine sofort wieder ein IRQ ausgelöst würde, da das 2. Bit nicht gelöscht ist. Dies muß daher unbedingt vom Programmierer in der IRQ-Routine vollzogen werden. Das Löschen geschieht, indem man das Register einfach ausliest und wieder zurückschreibt:

```
LDA IRR
STA IRR
```

Damit ist man auch in der Lage festzustellen, durch welches Ereignis der IRQ ausgelöst wurde. Dies ist wichtig, da anders als beim NMI ja immer noch ein Ereignis zusätzlich vorhanden ist, der Systeminterrupt. Da alle IRQs den gleichen Vektor benutzen, muß man durch Testen der einzelnen Bits des IRR feststellen, wodurch der IRQ ausgelöst wurde, sonst kann es z.B. passieren, daß eine Explosionsroutine, die für eine Sprite-Kollision vorgesehen war, plötzlich bei einem Systeminterrupt ausgelöst würde, was sicherlich nicht sehr angenehm für den Programmierer wäre.

Registerbelegung des VIC

Register 0: Sprite 0 X-Koordinate (Bits 0–7)

Register 1: Sprite 0 Y-Koordinate

Die Register 2–15 entsprechen im Aufbau den vorhergehenden und sind analog für die Sprites 1–7 zuständig.

Register 16: MSBs der X-Koordinaten. Die einzelnen Bits werden dabei den jeweiligen Sprites zugeordnet.

Register 17: Steuerregister 1
Bits 0–2 Vertikales Scrolling in Rasterzeilen
Bit 3 0=24 Zeilen, 1=25 Zeilen
Bit 4 0=Bildschirm aus,1=Bildschirm ein
Bit 5 0=Textmodus, 1=Grafikmodus
Bit 6 Extended-Color-Modus aus (0), ein (1)
Bit 7 8. Bit von Register 18

Register 18: Zugriff: Lesen
Nummer der Rasterzeile, die z.Z. aufgebaut wird.
Zugriff: Schreiben
Nummer der Rasterzeile, bei der ein IRQ ausgelöst wird.

Register 19: X-Koordinate des Lightpens nach einem Impuls.

Register 20: Y-Koordinate des Lightpens nach einem Impuls.

Register 21: Jedes Bit ist dem jeweiligen Sprite zugeordnet. Ist das Bit gesetzt, wird das Sprite aktiviert.

Register 22: Steuerregister 2
Bits 0–2 Vertikales Scrolling in Rasterpunkten
Bit 3 0=38 Spalten, 1=40 Spalten
Bit 4 Multi-Color-Modus ein (1), aus (0)

Register 23: Jedem Sprite wird das entsprechende Bit zugeordnet. Ist das Bit gesetzt, wird das Sprite vertikal expandiert.

Register 24: Basisadressen
Bits 1–3 Adreßbits 11–13 des Zeichengenerators
Bits 4–7 Adreßbits 10–13 des Video-RAM
Im Grafikmodus:
Bit 3 1=Grafikspeicher in den oberen 8 Kbyte
0=Grafikspeicher in den unteren 8 Kbyte des 16-Kbyte-Adreßraumes.

Register 25: Interrupt-Request-Register (IRR)
Bit 0 IRQ-Auslöser ist Register 18
Bit 1 IRQ-Auslöser ist Register 31
Bit 2 IRQ-Auslöser ist Register 30
Bit 3 IRQ-Auslöser ist der Lightpen
Bit 7 ist 1, wenn eins der Bits 0–3 gesetzt ist.

Register 26: Interrupt-Mask-Register (IMR)
Belegung wie Register 25. Ist ein Bit im IRR und IMR gesetzt, wird ein IRQ ausgelöst.

Register 27: 0: Sprite liegt vor Grafik
1: Sprite liegt hinter Grafik

Register 28: Jedem Sprite ist das entsprechende Bit zugeordnet. Ist dieses gesetzt, wird das Sprite im Multi-Color-Modus dargestellt.

Register 29: Wie Register 23, nur für horizontale Vergrößerung.

Register 30: Jedem Sprite ist das entsprechende Bit zugeordnet. Bei einer Kollision zwischen zwei Sprites werden die entsprechenden Bits und das Bit 2 des IRR gesetzt.

Register 31: Bei einer Sprite-Hintergrund-Kollision werden das dem Sprite zugeordnete Bit und das Bit 1 im IRR gesetzt.

Register 32: Rahmenfarbe.

Register 33–36: Hintergrundfarben.

Register 37–38: Multicolorfarben der Sprites.

Register 39–46: Farben der Sprites 0–7.

2.3.3.1 Der Rasterzeileninterrupt als IRQ-Auslöser

Wie oben schon erwähnt wurde, ist der VIC u.a. dafür zuständig, ein Bild auf dem Fernseher oder Monitor zu erzeugen. Um dies zu verstehen, muß an dieser Stelle (leider) ein wenig Physik eingefügt werden: Bei dem Bildschirm handelt es sich um eine phosphorisierte Schicht, auf die ein Elektronenstrahl geworfen wird. Durch den Aufprall von Elektronen beginnt diese Schicht punktweise zu leuchten. Der Elektronenstrahl, der in dem Monitor erzeugt wird, kann nun durch horizontale sowie vertikale Ablenkplatten beliebig auf den Bildschirm gesteuert werden. Die Richtung der Platten wird dabei durch das Signal an der Eingangsbuchse des Monitors erzeugt. Dieses Signal wird beim Fernseher normalerweise von den Rundfunkanstalten geliefert, wenn Sie jedoch Ihren C64 angeschlossen haben, übernimmt der VIC diese Aufgabe.

Dies alles geschieht mit einer ungeheuren Geschwindigkeit: Pro Sekunde wird vom VIC 20mal(!) ein neues Bild erzeugt, Sie können sich daher sicherlich ausmalen, wie der Elektronenstrahl über den Bildschirm rast (beim Fernsehen werden sogar 25 Bilder pro Sekunde aufgebaut!). Dieser wird vom Videocontroller in sogenannte Rasterzeilen und Rasterspalten eingeteilt, so daß jeder Bildpunkt praktisch eine Koordinate besitzt, durch die man ihn ansteuern kann. Im Bild anschließend an Kapitel 2.3.3.3. sehen Sie die genaue Einteilung in Zeilen und Spalten, wobei auffällt, daß die Auflösung des Textfensters der Punktauflösung bei der Multi-Color-Grafik entspricht (160 x 200 Punkte).

In diese Vorgänge kann man nun softwaremäßig eingreifen: Zum einen ist es möglich, durch Auslesen des VIC-Registers 18 die Rasterzeile festzustellen, die gerade vom VIC aufgebaut wird. Da man jedoch nur 8 Bit zur Verfügung hat, also maximal Werte bis 255 darstellen kann, wird das fehlende 9. Bit durch das 7. Bit des VIC-Registers 17 zur Verfügung gestellt. Der Haken bei der Sache ist jedoch, daß eine Rasterzeile in 178 Mikrosekunden aufgebaut wird, dies sind ca. 175 Taktzyklen. Wenn man bedenkt, daß ein Assemblerbefehl mindestens 2 Taktzyklen verbraucht, die meisten jedoch mehr, kann man sich vorstellen, daß man praktisch ständig abfragen müßte, welche Rasterzeile gerade aufgebaut wird. Von Basic aus ist diese Abfrage natürlich überhaupt nicht realisierbar. Die Rasterspalte können wir nicht irgendwo auslesen, dies ist aber auch gar nicht möglich, da jeder Punkt in einer Zeit von weniger als 1 Mikrosekunde aufgebaut wird und damit auch nicht durch die an sich extrem schnelle Maschinensprache behandelt werden kann.

Daher stellt uns der VIC die Möglichkeit zur Verfügung, beim Aufbau einer bestimmten Rasterzeile einen IRQ auszulösen. Dafür brauchen wir nur die Rasterzeile, bei der der IRQ ausgelöst werden soll, in das Register 18 (Lowbyte) bzw. in das 7. Bit des Registers 17 (Highbyte) zu schreiben und den IRQ durch Rasterzeilen wie oben beschrieben zu erlauben, indem wir die Bits 0 und 7 des IMR setzen. Von diesem Zeitpunkt ab wird jedesmal, wenn die von uns bestimmte Rasterzeile aufgebaut wird, ein IRQ ausgelöst, mit dem man geradezu verblüffende Effekte erzielen kann: Ein sehr beliebtes Anwendungsbeispiel stellt der »geteilte« Bildschirm dar, der auch unser Demothema sein soll. Dabei wird der Zeichensatz nach Erreichen einer bestimmten Rasterzeile wechselweise zwischen Groß-und Kleinschriftmodus umgeschaltet.

Dadurch wird in der oberen Bildschirmzone ein anderer Zeichensatz dargestellt als in der unteren. Denkbar wäre z.B. auch der Wechsel auf einen hochauflösenden Grafikschirm, etc. Auf diese Weise kann man sogar 16, 24 oder gleich 32 Sprites darstellen, indem in jedem Bildschirmabschnitt neue 8 Sprites aktiviert werden. Hier jedoch der Quelltext unseres Programms:

Listing: »rasterzeilen-irq«

```
c000          -;rasterzeilen-interrupt
c000          -         .ba $c000
006a          -         .eq rando    =    106
00c2          -         .eq randu    =    194
ea31          -         .eq irqalt   =    $ea31
d012          -         .eq raster   =    $d012
d01a          -         .eq mask     =    $d01a
d019          -         .eq request  =    $d019
d018          -         .eq modus    =    $d018
0015          -         .eq klein    =    21
0017          -         .eq gross    =    23
              -;
              -;initialisierung
              -;===============
              -;
c000 78       -         sei               ;interrrupt
c001 a9 1f    -         lda  #(irqneu)    ;verhindern
c003 a2 c0    -         ldx  #(irqneu)
c005 8d 14 03-          sta  $0314        ;irq-vektor auf
c008 8e 15 03-          stx  $0315        ;neue routine
c00b a9 6a    -         lda  #rando
c00d 8d 12 d0-          sta  raster       ;1.zeile fuer irq
c010 ad 11 d0-          lda  raster-1
c013 29 7f    -         and  #%01111111;highbyte loeschen
c015 8d 11 d0-          sta  raster-1
c018 a9 81    -         lda  #%10000001;irq durch raster-
c01a 8d 1a d0-          sta  mask         ;zeilen festlegen
c01d 58       -         cli               ;irq freigeben
c01e 60       -         rts
              -;
              -;neue interruptroutine
              -;=====================
              -;
c01f ad 19 d0-irqneu    lda  request      ;irq-register
c022 8d 19 d0-          sta  request      ;loeschen
c025 30 07    -         bmi  raster       ;zum raster - irq
              -;
              -;system-interrupt
              -;================
```

```
50:                   -;
51:   c027 ad 0d dc-        lda  $dc0d       ;irq-reg. loeschen
52:   c02a 58      -        cli              ;irq zulassen
53:   c02b 4c 31 ea-        jmp  irqalt      ;timer-irq-routine
54:                   -;
55:                   -;rasterzeilen-interrupt
56:                   -;=====================
57:                   -;
58:   c02e ad 12 d0-raster  lda  raster      ;zeile holen
59:   c031 c9 c2   -        cmp  #randu      ;unterer rand
60:   c033 b0 0a   -        bcs  ok          ;ja, sprung
61:   c035 a9 15   -        lda  #klein      ;nein, auf klein-
62:   c037 8d 18 d0-        sta  modus       ;schrift schalten
63:   c03a a9 c2   -        lda  #randu
64:   c03c 4c 46 c0-        jmp  exit        ;zum schluss
65:   c03f a9 17   -ok      lda  #gross      ;grossschriftmodus
66:   c041 8d 18 d0-        sta  modus       ;einschalten
67:   c044 a9 6a   -        lda  #rando
68:   c046 8d 12 d0-exit    sta  raster
69:   c049 4c 7e ea-        jmp  $ea7e       ;irq beenden
```

In der Initialisierungsroutine wird der IRQ-Vektor verbogen und danach die Rasterzeile für den ersten IRQ festgelegt. Anschließend wird der IRQ durch Rasterzeilen durch Beschreiben des IMR erlaubt. Die neue IRQ-Routine löscht als erstes das IRR (s.o.) und prüft gleichzeitig, ob der IRQ wirklich durch den VIC ausgelöst wurde. Dies erkennen wir an dem gesetzten Negativ-Flag, welches nämlich immer gesetzt ist, wenn das 7. Bit des IRR eins ist. Ist das Negativ-Flag nicht gesetzt, stammt der IRQ nicht vom VIC, worauf das IRQ-Register der CIA 1 gelöscht, der IRQ freigegeben und zur alten IRQ-Routine gesprungen wird. Die letzten Befehle werden Ihnen wahrscheinlich ein bißchen merkwürdig vorkommen: Was hat die CIA 1 mit dem VIC zu tun und warum wird der IRQ innerhalb eines Interrupts freigegeben? Wir werden dieses noch ausführlich besprechen.

Die Rasterzeilen-Interrupt-Routine ab Zeile 58 wechselt den Zeichensatz und legt die Zeile, bei der ein IRQ ausgelöst werden soll, neu fest. Auf diese Weise erhalten wir einen dreigeteilten Bildschirm, wobei in dem ersten und letzten Drittel jeweils der Kleinschrift-, im mittleren Teil der Großschriftmodus aktiviert ist. Im Demoprogramm wird dieser Effekt durch den Programmteil »Rasterzeilen-IRQ« verdeutlicht. Der Aufruf dieser Routine erfolgt mit

```
SYS 49152
```

Der Rasterzeilen-IRQ ist eine der fantastischsten Möglichkeiten überhaupt, an die man mit »normaler« Programmierung gar nicht herankommt. Ein Musterbeispiel für die Anwendung dieser Möglichkeiten stellt das Programm »4 Pseudo-VICs« aus der Zeitschrift 64er (Ausgabe Januar 1985) dar. Durch dieses wirklich gelungene Programm können Sie den Bildschirm in 4 völlig unabhängige Zonen einteilen, was u.a. eine Darstellung von 32 Sprites erlaubt. Die Grundlage war jedoch die gleiche wie bei unserem kleinen Demoprogramm, nämlich der Rasterzeileninterrupt.

2.3.3.2 Die Sprite-Kollisionen als IRQ-Auslöser

Die nächste IRQ-Möglichkeit des VIC besteht darin, bei Kollisionen von Sprites untereinander oder mit einem Hintergrundzeichen einen solchen auszulösen. Dies ist die Grundlage fast aller Spiele, die mit Sprites operieren, da nur auf diese Weise prompt auf etwaige Kollisionen z.B. mit Farb- und Toneffekten reagiert werden kann. Wie oben schon erwähnt wurde, müssen Sie im IRR die Bits 2 und 7 setzen, wenn der IRQ durch Sprite-Kollisionen untereinander ausgelöst werden soll, während durch Setzen der Bits 1 und 7 eine Sprite-Hintergrundkollision als IRQ-Auslöser dient. Möchten Sie beide Möglichkeiten kombinieren, müssen Sie daher die Bits 1, 2 und 7 im IRR setzen. In unserem kleinen Beispielprogramm wird von dieser Möglichkeit Gebrauch gemacht: Es handelt sich dabei um zwei Ballonfahrer, die miteinander kollidieren, wobei der zweite Ballon explodiert (Farbeffekt) und daraufhin abstürzt. Der erste, heilgebliebene Ballon stößt jedoch auf dem Rückflug mit einem Haus zusammen, wonach auch er abstürzt. Ein zugegeben recht primitives Programm, was jedoch zeigt, worauf es ankommt. Die Initialisierungsroutine gleicht der des vorhergehenden Beispiels, wobei natürlich das IMR mit der Bitkombination für Sprite-Kollisionen geladen wird. Genau wie beim Raster-IRQ prüft unsere neue IRQ-Routine zunächst, ob der IRQ vom Systeminterrupt oder vom VIC ausgelöst wurde und löscht das IRR. Wurde der IRQ durch den Systeminterrupt ausgelöst, wird wie beim Raster-IRQ fortgefahren. Die Kollisionsroutine ab Zeile 66 sorgt durch Beschreiben der entsprechenden VIC-Register für die Abstürze der Ballone, wobei vorher ab Zeile 58 zwischen der Ursache Sprite-Sprite und Sprite-Hintergrundkollision unterschieden wurde.

Obwohl man diese IRQ-Form anders als beim Rasterzeilen-Interrupt auch durch normale Abfragen der Kollisionsregister ersetzen kann, ist sie dieser Methode vorzuziehen, da Speicherplatz gespart wird und das Programm durch Entfallen der Abfrage wesentlich schneller ablaufen kann.

Listing: »sprite-irq«

```
09:     c04d           -;sprite-interrupt
12:     c04d           -          .ba $c04d
13:     d000           -          .eq vic      =    $d000
15:     ea31           -          .eq irqalt   =    $ea31
17:     d01a           -          .eq mask     =    $d01a
18:     d019           -          .eq request  =    $d019
22:                    -;
23:                    -;initialisierung
24:                    -;===============
25:                    -;
26:     c04d 78        -          sei             ;interrrupt
27:     c04e a9 5f     -          lda  #(irqneu)  ;verhindern
28:     c050 a2 c0     -          ldx  #(irqneu)
29:     c052 8d 14 03-            sta  $0314      ;irq-vektor auf
30:     c055 8e 15 03-            stx  $0315      ;neue routine
36:     c058 a9 86     -          lda  #%10000110;irq durch sprite-
```

```
37:   c05a 8d 1a d0-          sta  mask       ;kollisionen
38:   c05d 58       -         cli             ;festsetzen
39:   c05e 60       -         rts
40:                 -;
41:                 -;neue interruptroutine
42:                 -;====================
43:                 -;
44:   c05f ad 19 d0-irqneu    lda  request    ;irq-register
45:   c062 8d 19 d0-          sta  request    ;loeschen
46:   c065 30 07    -         bmi  sprite     ;zum vic - irq
47:                 -;
48:                 -;system-interrupt
49:                 -;================
50:                 -;
51:   c067 ad 0d dc-          lda  $dc0d      ;irq-reg. loeschen
52:   c06a 58       -         cli             ;irq zulassen
53:   c06b 4c 31 ea-          jmp  irqalt     ;system-irq-routine
54:                 -;
55:                 -;sprite-interrupt
56:                 -;================
57:                 -;
58:   c06e ad 1f d0-sprite    lda  vic+31     ;sprite-hintergrund
59:   c071 d0 37    -         bne  back
60:                 -;
61:                 -;sprite-sprite kollision
62:                 -;======================
63:                 -;
66:   c073 a2 23    -         ldx  #35
67:   c075 a0 00    -l1       ldy  #00
68:   c077 98       -l2       tya
69:   c078 8d 28 d0-          sta  vic+39+1   ;sprite1 farbe
70:   c07b 49 0f    -         eor  #15
71:   c07d 8d 29 d0-          sta  vic+39+2   ;sprite2 farbe
72:   c080 c8       -         iny
73:   c081 d0 f4    -         bne  l2
74:   c083 ca       -         dex
75:   c084 d0 ef    -         bne  l1
76:   c086 ee 03 d0-l3        inc  vic+3      ;sprite1 absturz
77:   c089 a2 0d    -         ldx  #13
78:   c08b a0 00    -lp1      ldy  #00
79:   c08d c8       -lp2      iny
80:   c08e d0 fd    -         bne  lp2
81:   c090 ca       -         dex
82:   c091 d0 f8    -         bne  lp1
87:   c093 ad 03 d0-          lda  vic+3
88:   c096 c9 dc    -         cmp  #220
89:   c098 d0 ec    -         bne  l3
```

```
c09a ad 15 d0-          lda  vic+21
c09d 29 fd    -         and  #%11111101;sprite1 aus
c09f 8d 15 d0-          sta  vic+21
c0a2 a9 00    -         lda  #00
c0a4 8d 1e d0-          sta  vic+30      ;kollision loeschen
c0a7 4c bc fe-          jmp  $febc       ;irq beenden
              -;
              -;sprite-hintergrund kollision
              -;============================
              -;
c0aa a2 23    -back     ldx  #35
c0ac a0 00    -14       ldy  #00
c0ae 98       -15       tya
c0af 8d 29 d0-          sta  vic+39+2    ;sprite2 farbe
c0b2 c8       -         iny
c0b3 d0 f9    -         bne  15
c0b5 ca       -         dex
c0b6 d0 f4    -         bne  14
c0b8 ee 05 d0-16        inc  vic+5       ;sprite2 absturz
c0bb a2 0d    -         ldx  #13
c0bd a0 00    -lp3      ldy  #00
c0bf c8       -lp4      iny
c0c0 d0 fd    -         bne  lp4
c0c2 ca       -         dex
c0c3 d0 f8    -         bne  lp3
c0c5 ad 05 d0-          lda  vic+5
c0c8 c9 dc    -         cmp  #220
c0ca d0 ec    -         bne  16
c0cc ad 15 d0-          lda  vic+21
c0cf 29 fb    -         and  #%11111011;sprite2 aus
c0d1 8d 15 d0-          sta  vic+21
c0d4 a9 00    -         lda  #00
c0d6 8d 1f d0-          sta  vic+31      ;kollision loeschen
c0d9 4c bc fe-          jmp  $febc       ;irq beenden
```

Ein isolierter SYS-Aufruf dieser Routine ist nicht sinnvoll, da natürlich solange überhaupt nichts passiert, solange keine Sprites vorhanden sind, die sich über den Bildschirm bewegen. Wenn Sie aber ein Programm erstellt haben, in dem Sprites aktiviert werden, wie dies auch in dem Demo-Programm der Fall ist, können Sie die Routine zu Beginn mit

```
SYS 49229
```

aktivieren.

2.3.3.3 Impuls vom Lightpen/Joystick als IRQ-Auslöser

Die letzte IRQ-Möglichkeit des VIC besteht darin, bei einem externen Impuls durch einen an Control Port 1 angeschlossenen Lightpen einen IRQ auszulösen. Durch die identische Pinbelegung haben Sie aber auch die Möglichkeit, diesen Impuls durch den Feuerknopf eines Joysticks zu erzeugen.

Ein Lightpen ist ein Stift, mit dem Sie durch Auflegen seiner Spitze auf den Bildschirm dem Computer die Position dieses Punktes mitteilen können. Daraufhin könnten Sie beispielsweise diese Koordinaten vom Programm aus nutzen, indem Sie an der angegebenen Stelle einen Grafikpunkt setzen. Dadurch wären Sie in der Lage, »per Hand« auf dem Bildschirm zu zeichnen. Die Funktion des Lightpen ist recht einfach zu erklären: Wie Sie aus Kapitel 2.3.3.1. wissen, wird jeder Punkt des Bildschirms durch einen Elektronenstrahl zum Aufleuchten gebracht. Dieses Aufleuchten wird vom Lightpen registriert, der daraufhin einen Impuls an den Rechner sendet. Natürlich »weiß« der VIC jederzeit, welchen Punkt des Bildschirms er gerade aufbaut, so daß er sofort nach dem Erhalt des Impulses die aktuelle Rasterzeile und Rasterspalte speichern kann. Dies geschieht in den Registern 19 (Rasterspalte, X-Koordinate) und 20 (Rasterzeile, Y-Koordinate). Der Programmierer kann nun durch Auslesen dieser Register die Koordinaten auswerten.

Wie aus dem Bild im Anschluß an dieses Kapitel ersichtlich ist, beginnt das Textfenster erst bei Rasterspalte 50, so daß Sie von dem gelesenen Wert aus Register 19 immer 50 subtrahieren müssen. Zusätzlich müssen Sie diese Zahl noch verdoppeln, da Sie 320 Grafikpunkte in X-Richtung zur Verfügung haben, jedoch nur 160 Rasterspalten. In Y-Richtung stimmt die Grafik- und Rasterzeilen-Auflösung überein, so daß Sie hier nur jeweils 50 von dem aus Register 20 erhaltenen Wert subtrahieren müssen. Die endgültigen Gleichungen, wobei »x« und »y« die tatsächlichen und »xl« sowie »yl« die ausgelesenen Koordinaten darstellen, sehen daher folgendermaßen aus:

x = 2 * (xl–50) und y = yl–50.

Genau wie bei den Sprite-Kollisionen tritt hierbei ein großes Problem auf: Während des normalen Programmablaufes merken wir nicht, ob der VIC einen Impuls vom Lightpen erhalten hat. Wir müssen daher praktisch ständig die Register 19 und 20 überprüfen, ob in diesen die Koordinaten eines neuen Punktes gespeichert sind, um entsprechend reagieren zu können. Zum Glück erlaubt aber auch hier der VIC einen IRQ, so daß wir augenblicklich mit der Auswertung des Lightpen-Impulses beginnen können. Unser Beispielprogramm reagiert auf einen Impuls vom Lightpen bzw. Joystick mit wechselseitiger Umschaltung von Text- und Grafikschirm.

Listing: »lightpen-irq«

```
c0df      -;lightpen-interrupt
c0df      -          .ba $c0df
d000      -          .eq vic       =    $d000
009b      -          .eq flag      =    $9b
ea31      -          .eq irqalt    =    $ea31
```

```
        d01a          -          .eq mask     =     $d01a
        d019          -          .eq request  =     $d019
                      -;
                      -;initialisierung
                      -;===============
                      -;
        c0df 78       -          sei                ;interrrupt
        c0e0 a9 24    -          lda   #(irqneu) ;verhindern
        c0e2 a2 c1    -          ldx   #(irqneu)
        c0e4 8d 14 03-           sta   $0314        ;irq-vektor auf
        c0e7 8e 15 03-           stx   $0315        ;neue routine
        c0ea a9 00    -          lda   #00          ;flag fuer text
        c0ec 85 9b    -          sta   flag         ;setzen
        c0ee a9 00    -          lda   #($6000)
        c0f0 85 71    -          sta   $71
        c0f2 a9 60    -          lda   #($6000)     ;grafikschirm
        c0f4 85 72    -          sta   $72
        c0f6 a9 00    -          lda   #00          ;ab $6000
        c0f8 a2 20    -          ldx   #32
        c0fa a8       -11        tay                ;loeschen
        c0fb 91 71    -12        sta   ($71),y
        c0fd c8       -          iny
        c0fe d0 fb    -          bne   12
        c100 e6 72    -          inc   $72
        c102 ca       -          dex
        c103 d0 f5    -          bne   11
        c105 a9 00    -          lda   #($4400)     ;videoram ab
        c107 85 71    -          sta   $71
        c109 a9 44    -          lda   #($4400)     ;$4400 mit farbe
        c10b 85 72    -          sta   $72
        c10d a9 6e    -          lda   #110         ;fuellen, punkt-
        c10f a2 04    -          ldx   #04
        c111 a0 00    -13        ldy   #00          ;farbe hellblau,
        c113 91 71    -14        sta   ($71),y
        c115 c8       -          iny                ;hintergrund blau
        c116 d0 fb    -          bne   14
        c118 e6 72    -          inc   $72
        c11a ca       -          dex
        c11b d0 f4    -          bne   13
        c11d a9 88    -          lda   #%10001000;irq durch lightpen
        c11f 8d 1a d0-           sta   mask         ;oder joystick
        c122 58       -          cli                ;festlegen
        c123 60       -          rts
                      -;
                      -;neue interruptroutine
                      -;=====================
                      -;
```

```
c124 ad 19 d0-irqneu   lda  request    ;irq-register
c127 8d 19 d0-         sta  request    ;loeschen
c12a 30 07    -        bmi  lightpen   ;zum lightpen - irq
              -;
              -;system-interrupt
              -;================
              -;
c12c ad 0d dc-         lda  $dc0d      ;irq-reg. loeschen
c12f 58       -        cli             ;irq zulassen
c130 4c 31 ea-         jmp  irqalt     ;timer-irq-routine
              -;
              -;lightpen-interrupt
              -;==================
              -;
c133 a5 9b    -lightpen lda flag       ;hgr oder text
c135 f0 1b    -        beq  hgr        ;grafik einschalten
              -;
              -;auf textschirm schalten
              -;=======================
              -;
c137 a9 1b    -        lda  #%00011011
c139 8d 11 d0-         sta  vic+17     ;grafik ausschalten
c13c a9 c8    -        lda  #%11001000;multicolor
c13e 8d 16 d0-         sta  vic+22     ;ausschalten
c141 a9 15    -        lda  #%00010101;zeichensatz auf
c143 8d 18 d0-         sta  vic+24     ;grosschrift
c146 a9 97    -        lda  #%10010111;16 k-verschiebung
c148 8d 00 dd-         sta  $dd00      ;des adressraumes
c14b a9 00    -        lda  #00        ;flag auf hgr
c14d 85 9b    -        sta  flag       ;schalten
c14f 4c 7e ea-         jmp  $ea7e      ;irq beenden
              -;
              -;auf grafikschirm schalten
              -;=========================
              -;
c152 a9 bb    -hgr     lda  #%10111011
c154 8d 11 d0-         sta  vic+17     ;grafik einschalten
c157 a9 c8    -        lda  #%11001000
c159 8d 16 d0-         sta  vic+22     ;multicolor aus
c15c a9 1d    -        lda  #%00011101
c15e 8d 18 d0-         sta  vic+24     ;videoram nach 4400
c161 a9 96    -        lda  #%10010110;16k-verschiebung
c163 8d 00 dd-         sta  $dd00      ;des adressraumes
c166 a9 01    -        lda  #01        ;flag auf text
c168 85 9b    -        sta  flag       ;schalten
c16a 4c 7e ea-         jmp  $ea7e      ;irq beenden
```

Die Initialisierungsroutine kennen Sie im Prinzip schon von den vorhergehenden Beispielen. Hier hat sie jedoch noch die Aufgabe, die hochauflösende Grafik einzurichten. Wichtig ist nur die Zeile 47, in der im IMR die Bits 3 (für Lightpen-IRQ) und 7 gesetzt werden. Die neue IRQ-Routine gleicht denen der letzten Beispiele. Ab Zeile 91 sehen Sie die Routine, die den Umschaltvorgang zwischen Grafik- und Textschirm realisiert, indem die entsprechenden VIC-Register und das Register 0 der CIA 1 beeinflußt werden. Letzteres legt den Adreßraum für den VIC fest, der bekanntlich ja nur 16 Kbyte gleichzeitig umfassen kann. Im Demoprogramm wird zunächst eine Sinuslinie auf dem Grafikschirm gezeichnet, durch Betätigung des Feuerknopfes bzw. des Lightpen können Sie beliebig zwischen Text- und Grafikschirm hin- und herschalten. Starten können Sie die Routine mit

```
SYS 49375
```

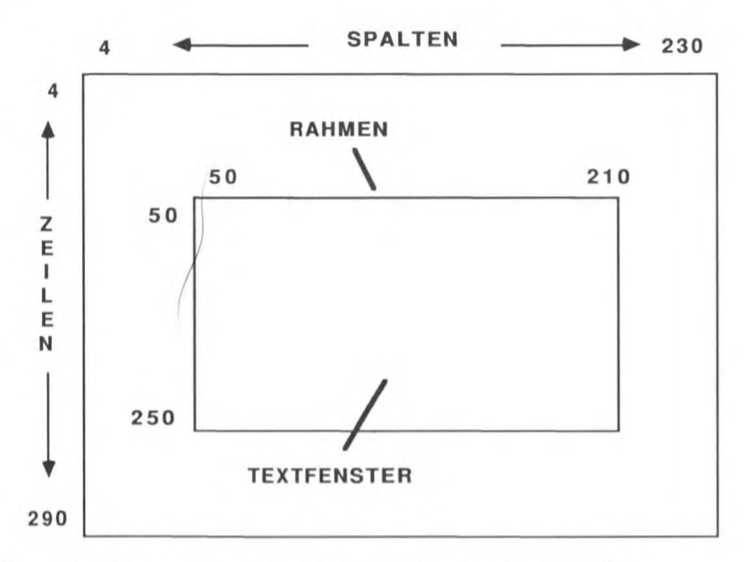


Bild 2.1: *Bildschirmkoordinaten durch Rasterzeilen und Rasterspalten*

2.3.4 Die Unterbrechung des IRQ durch einen IRQ

Wenn Sie die IRQ-Beispiele des VIC einmal vergleichen, sehen Sie, daß alle Routinen eine scheinbar merkwürdige Gemeinsamkeit aufweisen: Wenn sich herausstellte, daß der IRQ durch den Systeminterrupt ausgelöst wurde, wird das ICR der CIA 1 gelöscht und der IRQ mit dem CLI-Befehl freigegeben, obwohl wir uns in einer Interrupt-Routine befanden. An Hand des Rasterzeileninterrupts wollen wir dieses Rätsel lösen.

Wie in Abschnitt 1.3.3.1. erwähnt wurde, dauert der Aufbau eines kompletten Bildschirms ca. $^1/_{20}$ Sekunde. Da wir im Beispielprogramm während eines solchen Aufbaus zwei IRQs durch Rasterzeilen ausgelöst haben, bleibt zwischen je zwei IRQs eine Zeitspanne von ca. $^1/_{40}$ Sekunde. Da jedoch gleichzeitig der Systeminterrupt ca. alle $^1/_{60}$ Sekunde ausgelöst wird, lassen sich Überschneidungen der beiden IRQs auf Dauer nicht vermeiden, d.h., während ein Systeminterrupt abgearbeitet wird, wird irgendwann ein IRQ durch Rasterzeilen auftreten. Wir haben jedoch ganz zum Anfang des IRQ-Abschnitts gesehen, daß beim Auslösen eines IRQ automatisch das Interrupt-Flag gesetzt wird und somit weitere IRQs verhindert werden. Unser Rasterzeilen-Interrupt müßte also so lange warten, bis die Systeminterruptroutine beendet wäre. Da in dieser Zeit der Bildschirmaufbau jedoch schon fortgeschritten ist, wäre die Folge für unser Beispielprogramm eine unsaubere Trennung zwischen Klein- und Großschrift-Zeichensatz, die wir vermeiden wollen. Daher hat für uns der IRQ vom VIC höchste Priorität: Wir müssen auf jeden Fall einen IRQ auslösen, auch wenn gerade der Systeminterrupt behandelt wird. Das bedeutet, daß dieser System-IRQ durch unseren VIC-IRQ unterbrochen werden muß, um eine einwandfreie Funktion der Rasterroutine zu haben. Dies ist durchaus zulässig: Wird die normale IRQ-Routine unterbrochen, wird zunächst unsere VIC-Routine abgearbeitet und dann mit der Systeminterruptroutine fortgefahren. Wenn diese beendet ist, kann mit der Bearbeitung des ursprünglich unterbrochenen Programms weitergemacht werden. Dies klingt zwar kompliziert, ist aber im Prinzip sehr einfach und logisch. Durch das Lesen des ICR der CIA 1 müssen wir daher wie bei allen Interrupts die Ursache (Unterlauf Timer A (siehe Systeminterrupt)) löschen. Dann wird mit dem CLI-Befehl der IRQ freigegeben. Hätten wir vergessen, das ICR der CIA 1 zu löschen, würde jetzt sofort ein neuer IRQ ausgelöst, was ja nicht geschehen darf. Jetzt darf man die alte IRQ-Routine anspringen, da sie nun ja vom VIC wie gewünscht unterbrochen werden kann.

Damit wären wir am Ende der IRQ-Möglichkeiten durch den VIC und der gesamten IRQ-Programmierung angelangt. Wie wir jedoch gesehen haben, teilt sich der IRQ in den eigentlichen IRQ (den wir eben behandelt haben) und in die sogenannte BREAK-Routine auf. Obwohl diese softwaremäßig an sich kaum genutzt wird, soll sie nicht unerwähnt bleiben.

2.4 Die BREAK-Routine

Wie anfangs gesagt, wird durch das BREAK-Flag zwischen IRQ und BREAK unterschieden. Dieses Flag wird immer dann gesetzt, wenn in einem Maschinenprogramm der Assemblercode 00 (BRK) auftritt. Der BRK-Vektor liegt im RAM bei $0316 (Lowbyte) und $0317 (Highbyte). Daher können wir uns die Startadresse der BREAK-Routine mit

```
PRINT PEEK (790) + 256 * PEEK (791)
```

holen. Diese Startadresse liegt bei $FE66, einer Adresse, die wir schon kennen: Es handelt sich um einen Einsprung in die NMI-Routine, genauer gesagt wird der Teil durchlaufen, der bei einem NMI bei gedrückter RUN/STOP und RESTORE-Taste abgearbeitet wird. In

Kapitel 1 haben wir die NMI-Routine ausführlich besprochen, so daß ich mich hier nicht wiederholen möchte. Beim BREAK handelt es sich also um einen Programmabbruch, der in den Basic-Warmstart führt. Für uns ist der BREAK-Vektor daher uninteressant. Genutzt wird er bei fast allen Maschinensprachemonitoren, die bei einem BREAK im allgemeinen den Prozessorstatus (die Flags) anzeigen und in eine Eingabeschleife springen.

2.5 Der Abbruch eines IRQ durch den Programmierer

Zum Schluß habe ich noch einen kleinen Leckerbissen für Sie, ein Programm, das auf Tastendruck zu jedem beliebigen Zeitpunkt das Directory auf dem Bildschirm ausgibt. Dafür wird die Systeminterruptroutine benutzt. Das Problem besteht jedoch darin, daß man ein Programm, das auf die Floppy zugreift, nicht ablaufen lassen kann, ohne den Systeminterrupt abzuschalten, da die Floppy recht massiv in den Interruptvorgang eingreift. Im 64er Sonderheft »Assembler« ist eine solche Directoryroutine abgedruckt. Leider wird mit dem Systeminterrupt jedoch auch die Tastaturabfrage ausgeblendet. Unsere Directoryroutine soll jedoch den Komfort bieten, nach jedem gefüllten Bildschirm mit der weiteren Ausgabe so lange zu warten, bis der Anwender die RETURN-Taste betätigt, so daß keine Fileeinträge an ihm »vorbeirauschen«. Deshalb kann man auf die Tastaturabfrage hier nicht verzichten und damit den Systeminterrupt auch nicht während der Ausgabe abschalten. Nachdem also innerhalb des Interrupts festgestellt wurde, daß ein Directory erwünscht ist (Druck der £-Taste), müssen wir den Systeminterrupt verlassen und die eigentliche Directoryausgabe hauptprogrammäßig durchführen. Wie dies realisierbar ist, zeigt das

Listing: »abbruch-irq«

```
c1a5       -;abbruch-interrupt
c1a5       -         .ba $c1a5
009b       -         .eq zaehler  =   $9b
00cb       -         .eq taste    =   $cb
ea31       -         .eq irqalt   =   $ea31
00f9       -         .eq yr       =   $f9   ;y-register
00fa       -         .eq xr       =   $fa   ;x-register
00fb       -         .eq ac       =   $fb   ;accu
00fc       -         .eq st       =   $fc   ;status
00fd       -         .eq pzl      =   $fd   ;pcl
00fe       -         .eq pzh      =   $fe   ;pch
           -;
           -;initialisieren
           -;=============
           -;
c1a5 78    -         sei               ;interrupt verhindern
```

```
c1a6 a9 b2   -          lda  #(irqneu)
c1a8 a2 c1   -          ldx  #(irqneu)
c1aa 8d 14 03-          sta  $0314    ;irq-vektor auf neue
c1ad 8e 15 03-          stx  $0315    ;routine setzen
c1b0 58      -          cli           ;interrupt freigeben
c1b1 60      -          rts
             -;neue interruptroutine 1
             -;=====================
             -;
c1b2 a5 cb   -irqneu    lda  taste    ;letzte taste
c1b4 c9 30   -          cmp  #48      ;£
c1b6 f0 03   -          beq  dir      ;ja, directory
c1b8 4c 31 ea-          jmp  irqalt
             -;
             -;directory-interrupt
             -;==================
             -;
c1bb 68      -dir       pla           ;vom stapel holen
c1bc 85 f9   -          sta  yr
c1be 68      -          pla           ;y,x-register,
c1bf 85 fa   -          sta  xr
c1c1 68      -          pla           ;akku,status (flags),
c1c2 85 fb   -          sta  ac
c1c4 68      -          pla           ;ruecksprungadresse
c1c5 85 fc   -          sta  st
c1c7 68      -          pla           ;low und highbyte
c1c8 85 fd   -          sta  pzl
c1ca 68      -          pla
c1cb 85 fe   -          sta  pzh
c1cd a9 c1   -          lda  #(dirout)
c1cf 48      -          pha           ;auf stapel schieben
c1d0 a9 ec   -          lda  #(dirout)
c1d2 48      -          pha           ;neue ruecksprung-
c1d3 a5 fc   -          lda  st       ;adresse high- und
c1d5 48      -          pha
c1d6 a5 fb   -          lda  ac       ;lowbyte, status,
c1d8 48      -          pha
c1d9 a5 fa   -          lda  xr       ;akku, x,y-register
c1db 48      -          pha
c1dc a5 f9   -          lda  yr
c1de 48      -          pha
c1df a9 31   -          lda  #(irqalt)
c1e1 8d 14 03-          sta  $0314    ;irq auf alte routine
c1e4 a9 ea   -          lda  #(irqalt)
c1e6 8d 15 03-          sta  $0315    ;setzen
c1e9 4c 31 ea-          jmp  irqalt   ;zum normalen irq
             -;
```

```
                  -;directoryausgabe
                  -;================
                  -;
c1ec a9 93        -dirout   lda  #147     ;bildschirm
c1ee 20 d2 ff-              jsr  $ffd2    ;loeschen
c1f1 a9 01        -         lda  #01      ;log.filenummer
c1f3 a2 08        -         ldx  #08      ;geraeteadresse
c1f5 a0 00        -         ldy  #00      ;sekundaeradresse
c1f7 20 ba ff-              jsr  $ffba    ;setzen
c1fa a9 01        -         lda  #01      ;filename= "$"
c1fc a2 b9        -         ldx  #(name)  ;setzen
c1fe a0 c2        -         ldy  #(name)
c200 20 bd ff-              jsr  $ffbd
c203 20 c0 ff-              jsr  $ffc0    ;file oeffnen
c206 a2 01        -         ldx  #01      ;und als eingabe-
c208 20 c6 ff-              jsr  $ffc6    ;kanal setzen
c20b 20 5a c2-              jsr  diraus   ;directoryausgabe
c20e 20 cc ff-              jsr  $ffcc    ;kanaele zurueck
c211 a9 01        -         lda  #01
c213 20 c3 ff-              jsr  $ffc3    ;file schliessen
c216 a9 0d        -         lda  #13      ;return
c218 20 d2 ff-              jsr  $ffd2    ;ausgeben
c21b 20 e4 ff-16            jsr  $ffe4    ;auf return
c21e c9 0d        -         cmp  #13      ;von tastatur
c220 d0 f9        -         bne  16       ;warten
c222 a9 00        -         lda  #00      ;tastaturpuffer
c224 85 c6        -         sta  $c6      ;loeschen
c226 78           -         sei
c227 a9 35        -         lda  #(irqneu1) ;irq wieder
c229 a2 c2        -         ldx  #(irqneu1) ;auf 2. neue
c22b 8d 14 03-              sta  $0314    ;routine setzen
c22e 8e 15 03-              stx  $0315
c231 58           -         cli
c232 4c 32 c2-wait          jmp  wait     ;auf irq warten
                  -;
                  -;neue interruptroutine 2
                  -;======================
                  -;
c235 68           -irqneu1  pla           ;alles vom
c236 68           -         pla
c237 68           -         pla           ;stapel holen
c238 68           -         pla
c239 68           -         pla           ;s.o.
c23a 68           -         pla
c23b a5 fe        -         lda  pzh
c23d 48           -         pha           ;alte werte
c23e a5 fd        -         lda  pzl
```

```
c240 48        -          pha            ;auf stapel
c241 a5 fc     -          lda  st
c243 48        -          pha            ;schieben
c244 a5 fb     -          lda  ac
c246 48        -          pha            ;s.o.
c247 a5 fa     -          lda  xr
c249 48        -          pha
c24a a5 f9     -          lda  yr
c24c 48        -          pha
c24d a9 b2     -          lda  #(irqneu);irq-vektor
c24f a2 c1     -          ldx  #(irqneu)
c251 8d 14 03-            sta  $0314       ;auf 1.neue
c254 8e 15 03-            stx  $0315     ;routine setzen
c257 4c 7e ea-            jmp  $ea7e     ;irq beenden
               -;
               -;directory auf bildschirm
               -;======================
               -;
c25a 20 cf ff-diraus      jsr  $ffcf     ;startadr.
c25d 20 cf ff-            jsr  $ffcf     ;ueberlesen
c260 a9 00     -          lda  #00       ;zaehler fuer
c262 85 9b     -          sta  zaehler ;zeilen auf null
c264 20 cf ff-lp1         jsr  $ffcf     ;blockzeiger
c267 20 cf ff-            jsr  $ffcf     ;ueberlesen
c26a 20 cf ff-            jsr  $ffcf     ;blockzahl low
c26d 85 63     -          sta  $63       ;merken
c26f 20 cf ff-            jsr  $ffcf     ;blockzahl high
c272 85 62     -          sta  $62
c274 a2 90     -          ldx  #$90      ;in fliesskomma
c276 38        -          sec            ;wandeln, in
c277 20 49 bc-            jsr  $bc49     ;ausgabepuffer
c27a 20 dd bd-            jsr  $bddd     ;holen
c27d 20 1e ab-            jsr  $ab1e     ;und ausgeben
c280 20 cf ff-lp2         jsr  $ffcf     ;zeichen lesen
c283 a6 90     -          ldx  $90       ;datei ende
c285 f0 01     -          beq  weiter    ;nein, weiter
c287 60        -          rts            ;zurueck
c288 20 d2 ff-weiter      jsr  $ffd2     ;zeichen ausgabe
c28b c9 22     -          cmp  #$22      ;gaensefuesschen
c28d d0 f1     -          bne  lp2       ;nein
c28f 20 cf ff-lp3         jsr  $ffcf     ;zeichen lesen
c292 20 d2 ff-            jsr  $ffd2     ;und ausgeben
c295 c9 00     -          cmp  #00       ;zeilenende
c297 d0 f6     -          bne  lp3       ;nein
c299 a9 0d     -          lda  #13       ;return senden
c29b 20 d2 ff-            jsr  $ffd2
c29e e6 9b     -          inc  zaehler ;zaehler erhoehen
```

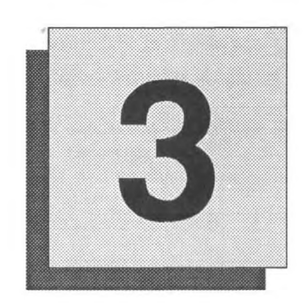
```
167:    c2a0 a5 9b    -         lda  zaehler ;bildschirm voll
168:    c2a2 c9 18    -         cmp  #24
169:    c2a4 d0 10    -         bne  ok      ;nein
170:    c2a6 a9 00    -         lda  #00
171:    c2a8 85 9b    -         sta  zaehler ;zaehler loeschen
172:    c2aa 85 c6    -         sta  $c6     ;tastaturpuffer=0
173:    c2ac 20 42 f1-lp4       jsr  $f142   ;auf taste warten
174:    c2af c9 0d    -         beq  lp4
175:    c2b1 a9 93    -         lda  #147    ;bildschirm
176:    c2b3 20 d2 ff-          jsr  $ffd2   ;loeschen
177:    c2b6 4c 64 c2-ok        jmp  lp1
178:                  -;
179:    c2b9 24       -name     .tx "$"      ;directoryfilename
```

Die Initialisierungsroutine kennen wir schon von unserem Beispiel, in dem wir den Cursor modifiziert haben. In der neuen IRQ-Routine wird dann geprüft, ob die »£«-Taste gedrückt wurde. Dies soll für uns das Zeichen sein, daß das Directory ausgegeben werden soll. Ist diese Taste nicht gedrückt, wird die alte IRQ-Routine bei $EA31 angesprungen. Von der Cursorroutine sind wir es gewohnt, erst unseren Job innerhalb des IRQs zu erledigen und dann die alte Routine anzuspringen. Dies ist jedoch bei Floppyzugriffen wie gesagt nicht erlaubt. Weil wir unsere Directoryroutine als Hauptprogramm ausführen wollen, müssen wir die Rücksprungadresse auf dem Stapel austauschen. Dafür müssen wir alles das herunterholen, was der Prozessor und die bisherige IRQ-Routine gestapelt haben. Zunächst werden daher Y- und X-Register, Akku, Prozessorstatus und die alte Rücksprungadresse geholt und gespeichert, da wir das alles noch brauchen werden (Zeilen 43–54). Anschließend schreiben wir die neue Rücksprungadresse auf den Stapel, die Anfangsadresse unserer Directoryroutine (Zeilen 55–58). Danach werden der alte Status, der Akku und die Register wieder gestapelt (Zeilen 59–66). Alles ist jetzt wie vorher, nur die ausgetauschte Rücksprungadresse zeugt von unseren Manipulationen. Anschließend wird der IRQ-Vektor wieder auf die alte Routine gesetzt (Zeilen 67–70) da wir während der Directoryausgabe keinen weiteren IRQ durch die »£«-Taste erlauben dürfen. Erst jetzt wird die alte IRQ-Routine angesprungen (Zeile 71).

Wenn der IRQ beendet wird, »denkt« der Prozessor durch die geänderte Rücksprungadresse, er wäre genau vor unserer Directoryroutine unterbrochen worden und springt diese daher unmittelbar an. Die Directoryroutine läuft von Zeile 76–99. Ist das Directory komplett ausgegeben, tritt für uns ein neues Problem auf: Wie kommen wir wieder in das alte, unterbrochene Programm zurück? Ein indirekter Sprung an die vorher gespeicherte Rücksprungadresse funktioniert nicht, da alle Register und der Status durch unsere Directoryausgabe verändert wurden. Daher machen wir genau das rückgängig, was wir getan haben, um in die Directoryroutine zu gelangen: Wir tauschen erneut die Rücksprungadresse auf dem Stapel. Deshalb wird der IRQ-Vektor auf die zweite neue Routine ab Zeile 111 »verbogen«. Anschließend wird in einer Warteschleife auf den nächsten Systeminterrupt gewartet. In der neuen IRQ-Routine wird der Stapel zunächst von allem »gesäubert«, was durch den Directoryinterrupt gestapelt wurde (Zeilen 111–116). Daraufhin werden alle Stapelwerte, die wir anfangs gespeichert hatten, also vom eigentlich unterbrochenen Programm stammen, auf den

Stack gepackt (Zeilen 117–128). Jetzt sind wir fertig: Auf dem Stapel sind genau die Anfangswerte und wir befinden uns wieder in der IRQ-Routine. Der Rechner kann daher nicht mehr zwischen dem Zustand vor der Directoryausgabe und nach ihr unterscheiden, wodurch eine ordnungsgemäße Weiterführung des unterbrochenen Programms gewährleistet ist! Zum Schluß müssen wir nur noch den IRQ-Vektor auf unsere ursprüngliche Routine setzen, die die »£«-Taste prüft, dann können wir mit einem Sprung den IRQ beenden. Es ist der gleiche Zustand eingetreten wie unmittelbar nach der Initialisierung. Durch SYS 49573 können Sie dieses schöne Programm aktivieren.

Sie sehen, daß man mit ein bißchen mehr Programmieraufwand auch Probleme lösen kann, die normalerweise nicht durch Interrupt behandelt werden dürfen. Die Directoryroutine ist ein Musterbeispiel hierfür. Durch Austausch der Directoryausgaberoutine gegen etwas anderes können Sie praktisch alles programmieren. Ein Beispiel wäre eine Routine, die ständig prüft, ob die Floppy eingeschaltet ist und entsprechend eine Fehlermeldung setzen kann. Möglich ist z.B. auch die Abspeicherung von ganzen Programmen durch den Systeminterrupt. Während des Speichervorgangs können Sie dabei anders als bei dem normalen »SAVE«-Befehl den Rechner weiter benutzen!



Variablen in Maschinensprache

Eines der größten Probleme stellt für den Maschinenprogrammierer die Organisation von Variablen dar. Als Basic-Benutzer kann man sich kaum vorstellen, wieviel Arbeit dazugehört, Variablen anzulegen und mit ihnen zu arbeiten. Es gibt daher nur ganz wenige professionelle Programme, die eine völlig eigene Variablenverwaltung aufweisen. Fast alle Programme benutzen jedoch die Struktur, wie sie auch der Basic-Interpreter verwendet. Damit ist es auch möglich, Interpreterroutinen zu benutzen. Natürlich unterliegt man, falls man diesen Weg geht, auch den Einschränkungen des Basic-Interpreters. Dieser läßt ja bekanntlich nur 4 Typen von Variablen zu:

- die Integervariablen
- die Fließkommavariablen
- die Strings
- die benutzerdefinierten Funktionen

Wenn man noch andere Typen in seinen Programmen verwenden möchte, muß man wohl oder übel eigene Strukturen entwerfen. Da man in der Regel jedoch mit den vier oben genannten Typen auskommt, kann man nur dazu raten, das Basic-Format auch in Maschinenprogramme zu übernehmen, um sich wirklich auf das Wesentliche konzentrieren zu können. Es gibt tatsächlich Programme, die an einer zu komplizierten Variablenverwaltung gescheitert sind, obwohl der Programmablauf an sich korrekt funktionierte. Daher wollen wir uns zunächst ansehen, wie der Interpreter die Variablen »bändigen« kann, wobei wir uns zunächst auf die nichtindizierten Variablen beschränken wollen. Die Feldvariablen (Arrays) wollen wir später behandeln.

3.1 Aufbau der nichtindizierten Variablen

3.1.1 Der Variablentyp INTEGER

Eine Integervariable besteht aus zwei Byte (16 Bit) und kann Zahlen im Bereich von –32768 bis +32767 darstellen. Das 7. Bit des höherwertigen Bytes stellt dabei das sogenannte Vor-

zeichenbit dar. Ist dies gesetzt, wird die Zahl als negativ interpretiert, so daß sich zwischen Dezimal- und Binärzahlen folgender Zusammenhang ergibt:

dezimal	binär	hexadezimal
–32768	1 000 0000 0000 0000	80 00
–32767	1 000 0000 0000 0001	80 01
–32766	1 000 0000 0000 0010	80 02
–32765	1 000 0000 0000 0011	80 03
–32764	1 000 0000 0000 0100	80 04
–2	1 111 1111 1111 1110	FF FE
–1	1 111 1111 1111 1111	FF FF
+–0	0 000 0000 0000 0000	00 00
+1	0 000 0000 0000 0001	00 01
+2	0 000 0000 0000 0010	00 02
+32763	0 111 1111 1111 1011	7F FB
+32764	0 111 1111 1111 1100	7F FC
+32765	0 111 1111 1111 1101	7F FD
+32766	0 111 1111 1111 1110	7F FE
+32767	0 111 1111 1111 1111	7F FF

Die Integerzahlen sind zur allgemeinen Verwendung kaum geeignet, da sie nur eine relativ geringe Bandbreite überdecken und zudem immer nur diskrete Werte ohne Nachkommastellen zulassen. Worin liegt aber die große Bedeutung dieses Variablentyps?

Die Antwort ist einfach: in der Geschwindigkeit der Rechenoperationen. Die Prozessoren des Typs 6502, 6510 usw. besitzen einige wenige Arithmetikbefehle, mit denen man in extrem kurzer Zeit Berechnungen der vier Grundrechenarten durchführen kann:

Addition, Subtraktion: `ADC, SBC.`
Multiplikation, Division: `ASL, ROL, LSR, ROR.`

Die vier letztgenannten Befehle bewirken eigentlich zwar immer nur eine Verschiebung eines Bytes um ein Bit, hierdurch wird der Wert jedoch verdoppelt oder halbiert. Damit kann man unmittelbar alle Multiplikationen und Divisionen um einen Faktor 2, 4, 8, 16 usw. durchführen. Durch geschickte Verkopplung mehrerer Zwischenrechnungen kann man jeden beliebigen Faktor erhalten, wie folgendes Beispiel zeigt:

Multiplikation einer Integerzahl in $FA/$FB mit dem Faktor 7:

Zahl*7 = Zahl*4 + Zahl*2 + Zahl*1.

```
LDX $FA         ;Lowbyte merken
LDY $FB         ;Highbyte merken
ASL $FA
ROL $FB         ;Zahl1 = Zahl*4
ASL $FA
ROL $FB
LDA $FB         ;Ergebnis merken
PHA
LDA $FA
PHA
STX $FA         ;Zahl wiederholen
STY $FB
ASL $FA         ;Zahl2 = Zahl*2
ROL $FB
PLA
CLC
ADC $FA         ;Zahl3 = Zahl1 +Zahl2
STA $FA
PLA
ADC $FB
STA $FB
TXA
CLC
ADC $FA         ;Ergebnis = Zahl3 + Zahl
STA $FA
TYA
ADC $FB
STA $FB
```

Die Integervariablen beim Commodore 64 belegen insgesamt 7 Byte und sind wie folgt aufgebaut:

Byte:	1.	2.	3.	4.	5.	6.	7.
	Erstes Zeichen Variablen-name	Zweites Zeichen Variablen-name	Highbyte Integer-wert	Lowbyte Integer-wert	0 Füllbyte	0 Füllbyte	0 Füllbyte

Man sieht sofort, daß eigentlich vier Byte ausreichend wären, die drei Nullbyte sind offenbar überflüssig. Der Grund für die Platzverschwendung liegt darin, daß die Integervariable in ihrem Aufbau der Fließkommavariable, zu der wir noch kommen werden, angeglichen wurde. Dadurch brauchen sich z.B. die Suchroutinen für eine Variable nur auf den Abstand von 7 Byte einzustellen. Das Erstaunliche jedoch besteht darin, daß der Basic-Interpreter praktisch keine

Routinen zur Rechnung mit Integervariablen besitzt. Vielmehr werden diese in Fließkommavariablen umgerechnet, bevor die eigentliche Rechenoperation beginnt. Anschließend wird dieser Wert wieder in das Integer-Format umgewandelt. Dieses Verfahren führt dazu, daß in der Praxis die Integervariablen von Basic geschwindigkeitsmäßig langsamer als Fließkommavariablen verarbeitet werden. Da keine Platzersparnis mit ihrer Verwendung verbunden ist, kann man dem Basic-Programmierer nur von ihrer Benutzung abraten. Der Maschinenprogrammierer jedoch kann durch eigene schnelle Routinen (s.o.) den Vorteil dieser Variablen voll zum Tragen bringen. Das gesetzte Bit 7 der beiden Buchstaben des Variablennamens dient zur Unterscheidung des Variablentyps von den übrigen drei. Durch diesen eleganten Weg konnte man ein eigenes Byte zur Unterscheidung der Variablen sparen, da das 7. Bit bei dem ASCII-Code eines Buchstabens nie gesetzt ist.

Neben den zeitkritischen Routinen stellt die Grafikprogrammierung beim Commodore 64 das Paradebeispiel für die Verwendung von Integerzahlen dar. Durch die Lage der Koordinaten von 0 bis 319 in X-Richtung und von 0 bis 199 in Y-Richtung kann man ohne Einschränkungen mit Integer-Variablen arbeiten, soweit man keine Werte wie Sinus, Cosinus usw. berechnen muß (z.B. bei Ellipsen). Um diesen Berechnungen zu entgehen, verwenden professionelle Grafikprogramme für die Kreisberechnung meist den Algorithmus X↑2+Y↑2=R↑2 (R:Radius des Kreises), um auch dort mit Integer-Variablen arbeiten zu können.

3.1.1.1 Rechnen mit Integerzahlen

Die möglichen Rechenoperationen mit Integerzahlen sind im Gegensatz zur Fließkommarechnung (s.u.) sehr beschränkt, da das Ergebnis ja nur dann wirklich sinnvoll ist, wenn es wieder eine Integerzahl darstellt. Es gibt nur vier Operationen, bei denen das Ergebnis zwangsläufig wieder eine Integerzahl ist:

- Addition
- Subtraktion
- Multiplikation
- Potenzierung (Sonderfall der Multiplikation)

Die Voraussetzung ist natürlich die, daß sich das Ergebnis im Bereich von –32768 bis +32767 bewegt. Als Sonderfall kann man u.U. auch noch die Division betrachten, die jedoch nur in seltenen Fällen eine Integerzahl als Ergebnis liefert. Wir wollen uns daher auf die ersten drei Fälle beschränken.

Wie schon erwähnt, stellt der Basic-Interpreter kaum Rechenroutinen zur Verfügung. Die einzige für uns nutzbare Routine stellt die Multiplikation zweier Integerzahlen dar. Damit ist aber auch die schwierigste Rechenoperation erschlagen worden. Mit einer Additions- bzw. Subtraktionsroutine möchte ich Sie hier nicht langweilen, so daß wir sofort zur Multiplikation übergehen können.

a) Multiplikation zweier Integerzahlen (a*b)

Die Multiplikationsroutine des Basic-Interpreters wird normalerweise dazu benutzt, die Position eines Feldelements (Array) zu berechnen, wird von uns also völlig zweckentfremdet. Die Zahl a wollen wir in $FA/$FB und die Zahl b in $FC/$FD gespeichert annehmen.

Vor dem Aufruf der Routine müssen wir den ersten Faktor (a) in den Speicherstellen $71 und $72 bereitstellen, während vom zweiten Faktor die Speicheradresse ($FC/$FD) in die Adressen $5F und $60 übergeben werden muß. Zusätzlich muß das Y-Register mit dem Wert 1 belegt sein. Der Grund ist, daß der zweite Faktor durch indirekt-indizierte Adressierung (mit dem Y-Register als Index) geholt wird. Dann kann man die Multiplikationsroutine ab $B34C aufrufen. Das Ergebnis steht im Akku (Highbyte) sowie im X-Register (Lowbyte) zur Verfügung und soll wiederum nach $FA/$FB gespeichert werden:

```
LDA $FA      ;Lowbyte des ersten Faktors
STA $71      ;uebergeben
LDA $FB      ;Highbyte des ersten Faktors
STA $72      ;uebergeben
LDA #$00     ;Startadresse des zweiten Faktors
STA $5F      ;Lowbyte und
LDA #$FC     ;Highbyte
STA $60      ;uebergeben
LDY #$01     ;Y-Register mit Index 1 laden
JSR $B34C    ;Multiplikationsroutine aufrufen
STX $FA      ;Ergebnis Lowbyte
STA $FB      ;Ergebnis Highbyte
```

b) Vorzeichenwechsel einer Integerzahl

Neben den oben genannten Rechenoperationen ist besonders der Vorzeichenwechsel einer Zahl interessant. Dieses Problem ist nicht einfach dadurch zu lösen, daß man das Vorzeichenbit umdreht, da ja die positiven Zahlen von Null aufwärts gezählt werden, die negativen Zahlen jedoch abwärts. Während also das Lowbyte immer positiv anzusehen ist, ist das Highbyte genau dann negativ, wenn es größer oder gleich 128 ist (Bit 7 gesetzt). Einen Vorzeichenwechsel erreicht man dadurch, daß man sowohl Low- als auch Highbyte invertiert und das Ergebnis noch um den Wert eins erhöht. Die Invertierung kann jeweils mit dem Befehl

```
EOR #$FF
```

geschehen, wie folgendes Beispiel zeigt:

```
-32766 =  1000 0000 0000 0010
EOR #$FF  1111 1111 1111 1111
          0111 1111 1111 1101
+ 1       0000 0000 0000 0001
          0111 1111 1111 1110 = +32766
```

Im Maschinencode müssen wir natürlich das Low- und Highbyte jeweils getrennt behandeln, da unser Prozessor keine 16-Bit-Operationen zuläßt. Dazu soll das Vorzeichen einer Zahl in $FA/$FB umgedreht werden, das Ergebnis soll wieder in diesen Speicherstellen abgelegt werden:

```
   LDA $FA    ;Lowbyte
   EOR #$FF   ;invertieren
   TAX
   LDA $FB    ;Highbyte
   EOR #$FF   ;invertieren
   TAY
   INX        ;Lowbyte + 1
   BNE L1     ;kein Ueberlauf
   INY        ;Highbyte + 1
L1 STX $FA    ;Ergebnis Lowbyte
   STY $FB    ;und Highbyte speichern
```

3.1.1.2 Bildschirmausgabe einer Integerzahl

Um eine 2-Byte-Zahl auf dem Bildschirm auszugeben, besitzt der Basic-Interpreter eine Routine ab $BDCD. Das Problem besteht nur darin, daß die 16-Bit-Zahl als vorzeichenlos positiv angesehen wird, so daß also immer ein Ergebnis zwischen 0 und 65535 erscheinen wird.

Um mit dieser Routine auch negative Integerzahlen ausgeben zu können, muß man daher wie folgt vorgehen: Zunächst prüft man, ob die Zahl positiv ist. Wenn dies der Fall ist, kann die Ausgaberoutine sofort angesprungen werden. Ist die Zahl negativ, geben wir ein Minuszeichen aus und drehen dann das Vorzeichen um, wie wir es oben schon getan haben. Dann kann die Ausgaberoutine angesprungen werden. Vor dem Aufruf muß man das X-Register mit dem Lowbyte und den Akku mit dem Highbyte der Zahl laden. Wir wollen nun den Wert einer Integervariablen, deren Startadresse in den Speicherstellen $5F und $60 abgelegt ist, auf dem Bildschirm ausgeben:

```
   LDY #$02      ;Pointer auf Variablenwert (Highbyte)
   LDA ($5F),Y   ;Highbyte holen
   BMI L2        ;Bit 7 gesetzt, daher Zahl negativ
   TAX
   INY           ;Pointer auf Variablenwert (Lowbyte)
   LDA ($5F),Y   ;Lowbyte holen
   TAY
   JMP L1
L2 PHA           ;Highbyte merken
   LDA "-"       ;Minuszeichen
   JSR $FFD2     ;ausgeben
   PLA           ;Highbyte holen
   EOR #$FF      ;invertieren
   TAX           ;merken
```

```
    LDY #$03       ;Pointer auf Variablenwert (Lowbyte)
    LDA ($5F),Y    ;Lowbyte holen
    EOR #$FF       ;invertieren
    TAY
    INY            ;Lowbyte plus 1
    BNE L1         ;kein Ueberlauf
    INX            ;Highbyte plus 1
L1  TXA
    PHA            ;Lowbyte nach X
    TYA            ;und Highbyte nach A
    TAX            ;transferieren
    PLA
    JSR $BDCD      ;Variablenwert ausgeben
```

Damit sind wir auch schon am Ende der Integer-Zahlen angekommen und wenden uns dem nächsten Variablen-Typ zu, den Strings.

3.1.2 Der Variablentyp STRING

Eine Stringvariable kann man als Zeichenkette mit (fast) beliebigem Inhalt definieren. Von Basic aus sind bestimmte Zeichen wie " nicht zulässig, da man das Hochkomma dort auch als Kennzeichen für Anfang und Ende eines Strings verwendet. In Maschinensprache jedoch gelten diese Einschränkungen nicht. Eine Stringvariable ist wie folgt aufgebaut:

Byte:	1.	2.	3.	4.	5.	6.	7.
	Erstes Zeichen Variablen-name	Zweites Zeichen Variablen-name	Strich-länge String-descriptor	Adresse Low String-descriptor	Adresse High String-descriptor	0 Füllbyte	0 Füllbyte

Man erkennt, daß die Stringvariable den eigentlichen Inhalt, nämlich die Zeichenkette, nicht enthält, sondern nur einen Zeiger (Pointer) auf diese sowie eine Angabe über deren Länge (beides zusammen nennt man Stringdescriptor). Dies liegt daran, daß die Länge eines Strings im Gegensatz zu den anderen Variablentypen nicht festgelegt ist. Um eine einheitliche Länge aller Stringvariablen zu erhalten, mußte man daher diesen Weg beschreiten. Die maximale Länge ergibt sich durch die Verwendung eines Bytes (= 8 Bit) zu 255 Zeichen. Als Typkennzeichnung ist diesmal das 7. Bit nur des zweiten Buchstabens des Variablennamens gesetzt. Da die benötigte Länge einer Variablen nur 5 Byte beträgt, wurden die beiden restlichen Bytes wiederum mit Nullen aufgefüllt, um eine einheitliche Länge aller Variablen zu erhalten. Im Gegensatz zum Basic kann man mit Maschinensprache seine Zeichenketten natürlich auch an eigentlich nicht dafür vorgesehenen Plätzen plazieren, wie z.B. im Kassettenpuffer oder im

RAM unter dem Basic-ROM bzw. Kernal. In letzterem Fall muß man jedoch darauf achten, daß zunächst der Prozessorport umgeschaltet werden muß, bevor man auf diesen RAM-Bereich zugreifen kann.

3.1.2.1 Bildschirmausgabe einer Stringvariablen

Für die Stringausgabe besitzt der Interpreter eine Routine ab $AB25. Als Vorbereitung muß man die Adresse des Strings in die Zellen $22 (Lowbyte) und $23 (Highbyte) eintragen, die Stringlänge kommt ins X-Register. Die Startadresse unserer Variablen soll wieder in den Zellen $5F und $60 stehen.

```
LDY #$02        ;Pointer auf Stringlaenge
LDA ($5F),Y     ;Stringlaenge holen
TAX             ;und ins X-Register schieben
INY
LDA ($5F),Y     ;Stringpointer Lowbyte holen
STA $22
 INY
LDA ($5F),Y     ;Stringpointer Highbyte holen
STA $23
JSR $AB25       ;String ausgeben
```

3.1.3 Der Variablentyp FUNKTION

Als dritten Variablentyp läßt der C64 eine sogenannte »benutzerdefinierte Funktion« zu. Dieses ist ein Ausdruck des Formats, z.B. DEF FN F(X)=COS(X)↑2-SIN(X), wobei der Funktionsausdruck im Prinzip beliebig sein kann. Der Aufbau einer Funktionsvariablen sieht wie folgt aus:

Byte:	1.	2.	3.	4.	5.	6.	7.
	Erstes Zeichen +Bit 7	Zweites Zeichen	Lowbyte	Highbyte	Lowbyte	Highfcbyte	0
	Variablen-name	Variablen-name	Start-adresse des Funk-tionsaus-drucks	Start-adresse des Funk-tionsaus-drucks	Start-adresse der Funk-tions-variablen	Start-adresse der Funk-tions-variablen	Füllbyte

Wie alle anderen Variablentypen ist auch die Funktion genau sieben Byte lang, wobei nur sechs Byte benötigt werden. Als Kennzeichnung des Variablentyps ist diesmal das 7. Bit des ersten Buchstabens des Variablennamens gesetzt. Die Bytes 3 und 4 enthalten einen Zeiger auf den Funktionsausdruck im Basic-Text, in unserem Beispiel also auf den Ausdruck

COS(X)↑2-SIN(X). An dieser Stelle sieht man, daß man in Maschinensprache keine Funktionen benutzen kann, da man ja keinen Funktionsausdruck im Basic-Text besitzt! Die Bytes 5 und 6 bilden einen Zeiger auf die Funktionsvariable, hier also auf das »X«.

Um in Maschinensprache diesen Variablentyp zu benutzen, muß man ein Unterprogramm schreiben, das die Berechnungen nach der gewünschten Funktionsvorschrift vollzieht. Falls die Adresse der Funktionsvariablen im Programm konstant bleibt, kann sich das Unterprogramm am Anfang den Wert dieser holen. Falls die Variable wechselnde Adressen aufweist, ist es sinnvoll, die jeweilige Startadresse dem Unterprogramm beim Aufruf z.B. durch X- und Y-Register zu übergeben. Im Prinzip wird auch in Basic so verfahren: Die einzelnen Funktionsausdrücke werden in den dafür vorhandenen Routinen ausgerechnet und nacheinander zusammengezählt. Der Gewinn dieses Variablentypes für den Basic-Programmierer (weniger Schreibarbeit) ist für den Maschinenprogrammierer in diesem Sinne sowieso nicht vorhanden, so daß man von keinem echten Verlust sprechen kann.

3.1.4 Der Variablentyp FLIESSKOMMA

Die Menge aller auf dem C64 darstellbaren Zahlen nennt man Fließkommazahlen (den Namen werden wir noch genauer erklären). Dabei handelt es sich um einen sinnvollen Ausschnitt aus dem unendlich großen Zahlenspektrum, da man auf dem Rechner natürlich nicht beliebig große Zahlen darstellen kann. Wir werden jedoch noch sehen, daß dieser eigentlich winzige Ausschnitt normalerweise ausreicht, es sei denn, man rechnet mit Zahlen, wie sie Dagobert Duck zum Zählen seiner Geldmengen benötigt. Fließkommazahlen kann man nicht wie z.B. die Integerzahlen in ein bestimmtes Schema pressen. Eine Fließkommazahl kann auch eine Integerzahl sein, ebensogut jedoch auch eine positive oder negative gebrochene Zahl oder eine ganze Zahl, die außerhalb des Integer-Bereiches liegt. Eigentlich ist auch die Menge der Fließkommazahlen unendlich groß, da man für zwei beliebig dicht zusammenliegende Zahlen immer noch eine finden kann, die zwischen diesen beiden liegt. Da jedoch auch die Rechengenauigkeit eines Computers begrenzt ist (auch darauf kommen wir noch genauer zu sprechen), ist die Zahl der darstellbaren Zahlen doch eingeschränkt, wenn auch die Anzahl der möglichen Werte sehr groß ist. Daraus kann man schon erahnen, daß die Darstellung im Rechner nicht so einfach ist wie z.B. die der Integerzahlen oder der Strings. Deshalb haben wir sie auch bis zum Schluß für Sie aufgehoben.

Der Mensch rechnet im Gegensatz zum Computer normalerweise im Dezimalsystem, welches das Zahlensystem mit der Basis 10 ist. Wenn man eine bestimmte Zahl darstellen möchte, teilt man diese (wenn auch unbewußt) in die sogenannte Mantisse und den Exponenten zur Basis 10 auf. Die Mantisse ist dabei der Vorfaktor, der angibt, wie oft man die jeweilige Zehnerpotenz hernehmen muß, um die gewünschte Zahl zu erhalten:

7 = 7 * 10↑0
31 = 3.1 * 10↑1
1.111.000 = 1.111 * 10↑6

Man sieht jedoch, daß diese Zahlendarstellung nicht eindeutig ist; man kann ein und dieselbe Zahl auf beliebig viele Arten mit jeweils unterschiedlichem Exponenten darstellen:

31 = 0.31 * 10↑2 = 3.1 * 10↑1 = 31 * 10↑0 usw.

Deshalb hat man sich auf die sogenannte normalisierte Darstellung geeinigt. Dabei kann die Mantisse nur einen Wert zwischen eins und der Basis des Zahlensystems, hier also der 10, annehmen. In unserem Beispiel wäre also die 3.1 * 10↑1 die normalisierte Darstellung. An dieser Stelle erkennt man, warum die Zahlen »Fließkommazahlen« genannt werden: Durch Verschiebung des Kommas (dies kann durch alle Stellen der Mantisse »fließen«) und gleichzeitige Änderung des Exponenten kann man eine Zahl auf verschiedene Weisen darstellen. Das Komma hat also keinen festen Platz, sondern ändert diesen in Abhängigkeit der Zehnerpotenz. Bei einer Erhöhung des Exponenten um eins muß es um eine Stelle nach links verschoben werden, bei einer Verschiebung um eine Stelle nach rechts muß der Exponent um eins erniedrigt werden.

Mit ausschließlich positivem Exponenten kann man in der normalisierten Form nur Zahlen, die größer oder gleich eins sind, darstellen, da der kleinste Wert 1 * 10↑0 (10↑0 = 1 !) ist. Um kleinere Zahlen verwenden zu können, muß man daher auch negative Exponenten zulassen:

0.9 = 9 * 10↑–1

Negative Zahlen werden einfach durch ein Minussetzen der Mantisse erreicht, so daß man auf diese Weise tatsächlich das gesamte Zahlenspektrum abdecken kann. Durch Anwendung der Potenzgesetze kann man nun beginnen zu rechnen. Um zwei Dezimalzahlen beispielsweise zu multiplizieren, muß man die Mantissen multiplizieren und die Exponenten addieren:

3.5 * 12 = (3.5 * 10↑0) * (1.2 * 10↑1) = 4.2 * 10↑1 = 42

Bei der Addition kann man nur Zahlen mit gleichem Exponenten verwenden, so daß u.U. normalisierte Zahlen in eine andere Darstellung durch Verschiebung des Kommas der Mantisse umgewandelt werden müssen:

2.5 + 11 = (2.5 * 10↑0) + (1.1 * 10↑1)
= (0.25 * 10↑1) + (1.1 * 10↑1)
= 1.35 * 10↑1 = 13.5

Der entscheidende Vorteil dieser Darstellungsweise liegt darin, daß man jede Dezimalzahl in einem überschaubaren Format darstellen kann. Für den Computer ist es ja unerläßlich, daß jede Zahl, egal wieviele Stellen diese normalerweise besitzt oder wie groß sie ist, in einem bestimmten Format gespeichert werden kann, da eine Verarbeitung sonst unmöglich ist. Wir sehen also, daß wir im Dezimalsystem durch Verwendung von Mantisse und Exponent gute Ergebnisse erzielen können. Leider rechnet jedoch unser C64 im Binärsystem, also dem Zahlensystem mit der Basis zwei. Wir müssen nun sehen, wie wir diese Darstellungsweise auf das Binärsystem übertragen können.

Nun, versuchen wir doch einfach einmal, eine Binärzahl in Mantisse und Exponenten aufzuteilen:

1.1011 * 10↑1010 = 1.1011 * 10↑10 (binär)

= 1 * 10↑10 + 1 * 10↑9 + 0 * 10↑8 + 1 * 10↑7 + 1 * 10↑6
= 1024 + 512 + 0 + 128 + 64
= 1728

Auch gebrochene Binärzahlen sind unproblematisch:

1.101 * 10↑0 (binär)

= 1 * 10↑0 + 1 * 10↑–1 + 0 * 10↑–2 + 1 * 10↑–3
= 1 + 0.5 + 0 + 0.125
= 1.625

In der Praxis brauchen wir aber den umgekehrten Weg, d.h. wir müssen die uns vertrauten Dezimalzahlen in Binärzahlen umwandeln. Dazu gibt es ein Verfahren, das ich an einem Beispiel (1965,125) verdeutlichen möchte.

Zunächst wird die Zahl in den Vorkomma- (1965) und in den Nachkommaanteil (0.25) aufgeteilt. Der Vorkommaanteil wird mit Restbildung so lange durch zwei geteilt, bis die Null erreicht wird:

1965 : 2 =	982	Rest 1
982 : 2 =	491	Rest 0
491 : 2 =	245	Rest 1
245 : 2 =	122	Rest 1
122 : 2 =	61	Rest 0
61 : 2 =	30	Rest 1
30 : 2 =	15	Rest 0
15 : 2 =	7	Rest 1
7 : 2 =	3	Rest 1
3 : 2 =	1	Rest 1
1 : 2 =	0	Rest 1

Man erhält nun die Dezimalzahl einfach dadurch, daß man die Reste von unten nach oben als Binärziffern interpretiert:

11110101101 = 2↑10 + 2↑9 + 2↑8 + 2↑7 + 2↑5 + 2↑3 + 2↑2 + 2↑0
= 1024 + 512 + 256 + 128 + 32 + 8 + 4 + 1
= 1965 !!!

Der Nachkommaanteil wird genau umgedreht behandelt: Statt durch zwei zu dividieren, wird er so lange mit zwei multipliziert, bis der Nachkommaanteil des Ergebnisses Null wird. Die Vorkommastellen des Ergebnisses werden jedesmal abgeschnitten und notiert:

0.125 * 2 = 0.25 Vorkommastelle 0
0.25 * 2 = 0.5 Vorkommastelle 0
0.5 * 2 = 1.0 Vorkommastelle 1

Die gesuchte Binärzahl ergibt sich aus den Vorkommastellen von oben nach unten gelesen, hier also

0.001 = 0 * 2↑–1 + 0 * 2↑–2 + 1 * 2↑–3 = 0.125

Die Zahl 1965,125 wird also durch die Binärzahl 11110101101.001 repräsentiert. Da die Zahl 1965,125 auch als 1965,125 * 10↑0 geschrieben werden kann, also nicht normalisiert wurde, bevor die Umwandlung begann, ist auch unser Ergebnis auf den Exponenten Null bezogen: 11110101101.001=11110101101.001 * 2↑0.

Es ist aber auch im Binärsystem möglich, eine Normalisierung durchzuführen. Wie oben erwähnt, muß sich die Mantisse dabei im Bereich zwischen eins und der Basis des Zahlensystems, also zwei, bewegen. Da diese selbst jedoch ausgeschlossen bleibt, kann eine normalisierte Mantisse im Dualsystem nur eine Eins vor dem Komma haben, andere Werte sind nicht möglich. Dies ist, wie wir noch sehen werden, für die Computer-Darstellung sehr bedeutungsvoll. In unserem Beispiel wäre also die normalisierte Form der Zahl

1110101101001 * 2↑10

Interessant ist jedoch, daß es nicht immer möglich ist, eine Dezimalzahl in eine Dualzahl umzurechnen, z.B. 0,3. Nach obigem Verfahren geht es los:

0.3 * 2 = 0.6 Vorkommastelle 0
0.6 * 2 = 1.2 Vorkommastelle 1
0.2 * 2 = 0.4 Vorkommastelle 0
0.4 * 2 = 0.8 Vorkommastelle 0
0.8 * 2 = 1.6 Vorkommastelle 1

An dieser Stelle wiederholt sich der Vorgang ab der zweiten Zeile, es wird nie ein Nachkommaanteil Null erreicht. Die Dualzahl würde daher nur durch eine Periode

0.01001

angenähert werden können. Dies ist jedoch kein Phänomen der Umwandlung ins Dualsystem. Die Zahl $^1/_3$ kann z.B. im Dreiersystem ohne weiteres als 1 * 3↑–1 dargestellt werden, während die Umwandlung in das Dezimalsystem versagt:

$^1/_3$ = 0.333

Bevor wir nun auf den C64 zu sprechen kommen, müssen wir uns noch über Zahlen, die kleiner als eins sind, Gedanken machen, da hier offenbar wiederum ein negativer Exponent erforderlich ist. Da im Zweiersystem keine negativen Zahlen existieren, müssen wir uns wieder des Tricks mit dem Vorzeichenbit bedienen. Wenn wir für den Exponenten ein Byte, d.h. 8 Bit reservieren wollen, können wir das 7. Bit ähnlich wie bei den Integerzahlen als Vorzeichenbit interpretieren, so daß Exponenten von –128 bis +127 darstellbar sind. Hier erkennt man nun auch, daß der Exponent für den eingangs erwähnten Bereich der benutzbaren Zahlen verantwortlich ist. Bei Verwendung eines Bytes für den Exponenten kann man daher Zahlen

von 2↑–128 = 2.938 * 10↑–39 bis 2↑127 = 1.701 * 10↑38

darstellen, was in fast allen Anwendungsfällen genügen dürfte. Da die Entwickler des C64 auch dieser Meinung waren, wurde auf ein zweites Byte für den Exponenten verzichtet. Mit ihm hätte man dann Zahlen

von 2↑–32768 bis 2↑32767

berechnen können. Der C64 arbeitet jedoch ohne Vorzeichenbit, statt dessen addiert er zu jedem Exponenten einen Offset von 129 (nicht 128, wie oft zu lesen !!!) und interpretiert die so entstandene Zahl als positiv und vorzeichenlos. Damit werden die tatsächlichen Exponenten –128 bis 126 in die Zahlen 1 bis 255 umgewandelt. Die Zahl 127 kann in diesem Format nicht mehr berücksichtigt werden, während die Null unbenutzt bleibt. Der Grund hierfür liegt darin, daß bisher eine wichtige Zahl überhaupt noch nicht behandelt wurde: die Null! In der Exponentenschreibweise kommt man an sie nicht heran, so daß ihr vereinbarungsgemäß der Exponent Null zugeordnet wird:

Exponent	Darstellung	Wert
0	0	0
–128	1	2.93 * 10↑-39
–127	2	5.87 * 10↑-39
–126	3	1.17 * 10↑-38
–125	4	2.35 * 10↑-38
–2	127	0.25
–1	128	0.5
+–0	129	1
+1	130	2
+2	131	4
+122	251	5.32 * 10↑36
+123	252	1.06 * 10↑37
+124	253	2.12 * 10↑37
+125	254	4.25 * 10↑37
+126	255	8.50 * 10↑37

Nachdem wir also festgestellt haben, daß der Exponent für den Rechenbereich verantwortlich ist, kommen wir zu der Mantisse, die über die Rechengenauigkeit entscheidet. Der C64 verwendet für sie 4 Byte, d.h., daß 32 Binärzeichen darstellbar sind. Dabei wird linksbündig gearbeitet, d.h., alle nicht benötigten Bits werden mit Nullen aufgefüllt. Auf der anderen Seite bedeutet dies aber auch, daß Mantissen, die mehr als 32 Stellen aufweisen, nach dieser einfach abgeschnitten werden. Da das oberste Bit des ersten Bytes die Vorkommastelle der Mantisse darstellt, die ja immer eins ist (s.o.), ist das unterste Bit des vierten Bytes für den Wert 2↑–31 verantwortlich, z.B.

```
1. Byte      2. Byte      3. Byte      4. Byte

1 1110101    10011011     01010201     10011011
2 2222222222222222222222222222222222222222
↑ ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
0 ----------------------------------------------------------
  1234567    89111111     11112222     22222233
               012345     67890123      4567891
```

Damit kann man also Zahlen, die sich um den Wert 2↑–31 unterscheiden, gerade noch differenzieren. Dabei handelt es sich um den Wert

4.656612873 * 10↑–10 = 0.000 000 000 2

Der C64 besitzt also eine Genauigkeit von 9 Stellen, von der zehnten Stelle an wird ungenau gerechnet. Das Problem bei solchen Rundungsfehlern liegt darin, daß der Fehler mit der Anzahl der Rechenoperationen immer größer wird.

Bis auf ein Problem haben wir nun alles gelöst: Wie kann man ein negatives Mantissenvorzeichen darstellen? Nun, hier gibt es zwei Wege. Zum einen das sogenannte »Floating-Point«-Format, das das Vorzeichen in ein eigenes Byte verpackt. Als Kennzeichen wird das 7. Bit benutzt. Ist dies 1, wird die Zahl als negativ angenommen, bei einer Null als positiv. Die Bits 0–6 bleiben auf diese Weise unbenutzt. Da es sich hierbei um eine reine Platzverschwendung handelt, werden Fließkommavariablen im sogenannten »Memory-Floating-Point«-Format gespeichert. Hierbei wird ausgenutzt, daß das Vorkommabit im Dualsystem immer 1 ist. Da damit der Informationsgehalt des Bits Null ist, wird einfach dieses 7. Bit des ersten Mantissenbytes als Vorzeichenbit benutzt! Damit hat man eine platzsparende Variante gefunden, in der das 1-Bit also nicht mit abgespeichert, sondern durch das Vorzeichenbit ersetzt wird.

Nun haben wir alle Informationen beisammen, die uns den Aufbau der Fließkommavariablen im C64 verständlich machen:

Byte:	1.	2.	3.	4.	5.	6.	7.
	Erstes Zeichen	Zweites Zeichen	Exponent + 129	1. Bit 7: Vor-zeichen	2.	3.	4.
	Variablen-name	Variablen-name	≡	Mantissen-byte	Mantissen-byte	Mantissen-byte	Man-tissen-byte

Als Unterscheidungsmerkmal zu den übrigen Variablentypen wird hier das 7. Bit weder bei dem ersten noch bei dem zweiten Buchstaben des Variablennamens gesetzt. Die Fließkommavariable benötigt volle 7 Byte, wobei der Exponent vor den 4 Mantissenbyte steht. Man sieht also, daß die Nullbytes bei den Strings, Integers und Funktionen einzig und allein dazu dienten, die gleiche Länge wie die Fließkommavariablen zu erreichen.

Im Gegensatz zu den Integervariablen kann man mit Fließkommavariablen nur sehr schwer rechnen. Es bietet sich daher an, die im Basic-Interpreter vorhandenen Routinen auch für eigene Maschinenprogramme zu nutzen.

3.1.4.1 Rechnen mit Fließkommazahlen

Die Fließkomma-Akkumulatoren

Um mit Integerzahlen rechnen zu können, genügen meistens der Akku sowie das X- und Y-Register. Bei Fließkommazahlen sehen die Berechnungen jedoch ungleich komplizierter aus. Daher befinden sich im C64 zwei Fließkomma-Akkumulatoren (FAC), die sozusagen ein Rechenzentrum speziell für diese Zahlen darstellen. Der FAC I wird bei jeder Rechenoperation benutzt. Dazu wird die Zahl vor der Rechnung in ihm abgelegt, anschließend steht das Ergebnis zur Verfügung. Der FAC II dient dazu, bei Rechenoperationen, die zwei Zahlen erfordern, z.B. bei einer Addition, die zweite Zahl aufzunehmen. Das Ergebnis steht wieder im FAC I. Wir wollen den FAC I daher nur noch FAC nennen, den FAC II hingegen ARG (Argument). Im Gegensatz zu den Variablen werden die Zahlen hier im 6-Byte-Format (s.o.) abgespeichert, d.h., das Vorzeichen bekommt ein eigenes Byte, das 7. Bit des ersten Mantissenbytes wird wieder zu der Eins rekonstruiert. Daneben gibt es im FAC noch ein Rundungsbyte, welches bei diversen Operationen eine Rundung erlaubt, sowie ein Vorzeichenvergleichsbyte. Dieses gibt an, ob der FAC und ARG ein gleiches Vorzeichen ($00) oder ein unterschiedliches aufweisen ($FF). Die Fließkomma-Akkus benutzen die folgenden Speicherstellen:

	FAC	ARG
Exponent	$61	$69
1. Mantisse	$62	$6A
2. Mantisse	$63	$6B
3. Mantisse	$64	$6C
4. Mantisse	$65	$6D
Vorzeichen	$66	$6E
Rundung	$70	---
Vorzeichenvergleich	$6F	---

Bereitstellung einer Fließkommazahl

Um mit einer Fließkommazahl rechnen zu können, muß diese zunächst einmal im FAC bzw. ARG bereitstehen. Es gibt zwei Wege:

- Umwandlung einer ganzen Zahl ins Fließkommaformat mittels Interpreterroutinen.
- Laden des FAC oder ARG mit einer Fließkommavariablen aus dem RAM/ROM. Dabei ist es notwendig, diese Zahl von der 5-Byte- in die 6-Byte-Darstellung umzuwandeln.

Bereitstellung im FAC

Den FAC kann man auf beide der oben beschriebenen Arten mit einer Zahl versorgen.

Zunächst möchte ich auf die Umwandlung einer ganzen Zahl ins Fließkommaformat eingehen. Hierbei gibt es mehrere Möglichkeiten:

a) Ein-Byte-Wert mit Vorzeichen

Mit der folgenden Routine kann man Werte von –128 ($FF) bis +127 ($7F) umwandeln. Das Ergebnis wird im FAC abgelegt. Der Ein-Byte-Wert wird mit dem Akku übergeben. Der Einsprung geschieht in die SGN-Funktion ab $BC3C.

```
LDA #$Wert
JSR $BC3C
```

b) Ein-Byte-Wert ohne Vorzeichen

Möchte man einen Wert von 0 ($00) bis 255 ($FF) umwandeln, muß man die folgende Routine benutzen, die in die POS-Funktion des Interpreters einspringt:

```
LDY #$Wert
JSR $B3A2
```

c) Zwei-Byte-Wert mit Vorzeichen

Diese Routine dient dazu, gewöhnliche Integer-Zahlen im Bereich von –32768 ($80 $00) bis +32767 ($7F $FF) ins Fließkommaformat umzuwandeln. Während das Lowbyte im Y-Regi-

ster stehen muß, muß man den Akku mit dem Highbyte laden. Der Einsprung erfolgt in die FRE-Routine des Basic-Interpreters.

```
LDY #$LOW
LDA #$HIGH
JSR $B395
```

d) Zwei-Byte-Wert ohne Vorzeichen

Möchte man das Vorzeichen nicht berücksichtigen, kann man Zahlen von 0 ($00 $00) bis 65535 ($FF $FF) benutzen. Diesmal müssen die Werte direkt in den FAC übertragen werden, das Lowbyte an die Speicherstelle $63 und das Highbyte an die Stelle $62. Das X-Register wird mit dem Exponenten einschließlich Offset geladen. Da das höchste Bit bei 2-Byte-Werten 2^15 ist, muß man als Exponenten 15 + 129 = 144 = $90 ansetzen. Der SEC-Befehl dient dazu, ein Invertieren des FAC an der Stelle $B8D4 zu verhindern. Der Einsprung erfolgt wieder in die SGN-Funktion.

```
LDA #$HIGH
LDY #$LOW
STA $63
STY $62
SEC
LDX #$90
JSR $BC49
```

Da man in der Praxis kaum mit 3- oder gar 4-Byte-Werten in Maschinenprogrammen als ganze Zahlen arbeitet, möchte ich die entsprechenden Routinen zur Umwandlung hier nicht anführen. Wenn man, was die Regel ist, gebrochene oder extrem hohe Zahlen benötigt, wird man den zweiten Weg zur Umwandlung in Fließkommazahlen gehen, den wir nun besprechen werden.

Man wird nämlich die Zahl bereits im Fließkommaformat (5-Byte-Form) im RAM ablegen und dann den FAC mit einer speziellen Routine laden. Die Erzeugung einer Fließkommazahl im RAM kann man z.B. per Hand oder aber mit einem Assembler vornehmen. So bietet beispielsweise Profi-Ass 64 den Pseudo-Maschinenbefehl .FLP an. Dieser bewirkt, daß ab der assemblierten Stelle die nachfolgende Zahl im 5-Byte-Format (d.h. mit Vorzeichenbit in der Mantisse) abgelegt wird, z.B.

```
   .FLP 1  führt zu  $81 $00 $00 $00 $00
   .FLP 10 führt zu  $84 $20 $00 $00 $00
.FLP -8,25 führt zu  $84 $84 $00 $00 $00
```

Um diese Zahlen in den FAC zu übernehmen, gibt es eine Interpreterroutine, vor deren Aufruf man den Akku mit dem Lowbyte der Startadresse der Fließkommazahl und das Y-Register mit dem Highbyte laden muß. Angenommen, unsere Zahl stünde am Anfang des Kassettenpuffers (ab $033C), so müßte die Laderoutine wie folgt aussehen:

```
LDA #$3C  ;Lowbyte von $033C
LDY #$03  ;High-Startadresse
JSR $BBA2 ;Fließkommazahl in FAC übernehmen.
```

Bei dem Ladevorgang wird die Zahl automatisch vom 5-Byte-Format in das 6-Byte-Format für den FAC umgewandelt. Falls man eine Fließkommavariable in den FAC laden möchte, kennt man natürlich nur die Startadresse der Variablen. Wir erinnern uns, daß zunächst zwei Byte mit dem Variablennamen folgen, bevor die eigentlichen Daten kommen. Für den Ladevorgang einer Variablen ist daher folgende Routine geeignet, wobei wir davon ausgehen wollen, daß die Startadresse der Variablen in $5F/$60 zur Verfügung steht:

```
   LDY $60   ;Highbyte der Startadresse
   LDA $5F   ;Lowbyte derselben
   CLC
   ADC #02   ;+ 2 = Start der Daten
   BCC L1    ;Überlauf aufgetreten ?
   INY       ;Ja, Highbyte erhöhen
L1 JSR $BBA2 ;FAC mit Daten laden
```

Bereitstellung im ARG

Im allgemeinen wird der ARG sehr viel seltener gebraucht als der FAC. Man wird ihn daher praktisch nur auf die zweite Weise laden, indem man einen Fließkommawert aus dem RAM oder ROM einlädt. Der Weg ist genau analog zum FAC, der Unterschied besteht nur in der Einsprungadresse, auch hier wird das Lowbyte der Startadresse im Akku und das Highbyte im Y-Register übergeben:

```
LDA #$3C    ;Startadresse
LDY #$03    ;des Kassettenpuffers
JSR $BA8C   ;ARG mit Fließkommawert laden
```

Austausch der Fließkomma-Akkus

Es ist ohne weiteres möglich, die Inhalte von FAC und ARG zu tauschen bzw. zu übernehmen. Dafür existieren zwei Routinen:

```
ARG = FAC :  JSR $BC0C
FAC = ARG :  JSR $BBFC
```

Weiterhin kann man so den ARG z.B. als Zwischenspeicher verwenden, falls man einen Wert des FAC noch benötigt und dieser durch eine Rechenoperation verändert wird. Der interne Tausch geht schneller vor sich als die Zwischenspeicherung im RAM, da ja in letzterem Fall der Fließkommawert vom 5-Byte- ins 6-Byte-Format umgewandelt werden muß.

Die ROM-Konstanten

Der C64 hat im ROM eine stattliche Anzahl von Zahlen im 5-Byte-Fließkomma-Format gespeichert, die wir benutzen können, ohne sie im RAM anlegen zu müssen. Der Interpreter

braucht sie u.a. zur Berechnung von Logarithmen, usw. Wenn wir daher eine Zahl benötigen, die schon im ROM steht, brauchen wir diese nur einfach wie oben gesehen in den FAC einzuladen und können dann sofort mit ihr rechnen. Die folgende Tabelle zeigt die Konstanten und ihre Bedeutung auf:

Adresse	Fließkommawert	Dezimalwert	Bedeutung
$AEA8	82 49 0F DA A1	3.141592654	Pi
$B1A5	90 80 00 00 00	–32768	Zahl nach Integer wandeln
$B1BC	81 00 00 00 00	1	Konstante für LOG
$B1C2	7F 5E 56 CB 79	0.434255942	Konstante für LOG
$B9C7	80 13 9B 0B 64	0.576584541	Konstante für LOG
$B9CC	80 76 38 93 16	0.961800759	Konstante für LOG
$B9D1	82 38 AA 3B 20	2.88539007	Konstante für LOG
$B9D6	80 35 04 F3 34	0.707106781	1/SQR(2)
$B9DB	81 35 04 F3 34	1.414213562	SQR(2)
$B9E0	80 80 00 00 00	–0.5	Konstante für LOG
$B9E5	80 31 72 17 F8	0.693147181	LOG(2)
$BAF9	84 20 00 00 00	10	Berechnung FAC=FAC/10
$BDB3	9B 3E BC 1F FD	99999999.9	Berechnung FLP nach ASCII
$BDB8	9E 6E 6B 27 FD	999999999	Berechnung FLP nach ASCII
$BDBD	9E 6E 6B 28 00	1000000000	Berechnung FLP nach ASCII
$BF11	80 00 00 00 00	0.5	Konstante für SQR
$BFBF	81 38 AA 3B 29	1.44269504	1/LOG(2)
$BFC5	71 34 58 3E 56	2.149876E-5	Konstante für EXP
$BFCA	74 16 7E B3 1B	1.435231E-4	Konstante für EXP
$BFCF	77 2F EE E3 85	1.342263E-3	Konstante für EXP
$BFD4	7A 1D 84 1C 2A	9.614011E-3	Konstante für EXP
$BFD9	7C 63 59 58 0A	0.055505126	Konstante für EXP
$BFDE	7E 75 FD E7 C6	0.240226385	Konstante für EXP
$BFE3	80 31 72 18 10	0.693147168	Konstante für EXP
$BFE8	81 00 00 00 00	1	Konstante für EXP
$E08D	98 35 44 7A 00	11879546	Konstante für RND
$E092	68 28 B1 46 00	3.927677E-4	Konstante für RND
$E2E0	81 49 0F DA A2	1.57079633	PI/2
$E2E5	83 49 0F DA A2	6.28318531	PI*2
$E2EA	7F 00 00 00 00	0.25	Konstante für SIN/COS
$E2F0	84 E6 1A 2D 1B	–14.3813907	Konstante für SIN/COS
$E2F5	86 28 07 FB F8	42.0077971	Konstante für SIN/COS
$E2FA	87 99 68 89 01	–76.7041703	Konstante für SIN/COS
$E2FF	87 23 35 DF E1	81.6052237	Konstante für SIN/COS
$E304	86 A5 5D E7 28	–41.3147021	Konstante für SIN/COS

Fortsetzung

Adresse	Fließkommawert	Dezimalwert	Bedeutung
$E309	83 49 0F DA A2	6.28318531	PI*2
$E33F	76 B3 83 BD D3	–6.84793E-4	Konstante für ATN
$E344	79 1E F4 A6 F5	4.850942E-3	Konstante für ATN
$E349	7B 83 FC B0 10	–0.16111701	Konstante für ATN
$E34E	7C 0C 1F 67 CA	0.34209638	Konstante für ATN
$E353	7C DE 53 CB C1	–0.05427913	Konstante für ATN
$E358	7D 14 64 70 4C	0.072457196	Konstante für ATN
$E35D	7D B7 EA 51 7A	–0.08980191	Konstante für ATN
$E362	7D 63 30 88 7E	0.110932413	Konstante für ATN
$E367	7E 92 44 99 3A	–0.14283980	Konstante für ATN
$E36C	7E 4C CC 91 C7	0.19999912	Konstante für ATN
$E371	7F AA AA AA 13	–0.33333331	Konstante für ATN
$E376	81 00 00 00 00	1	Konstante für ATN
$E3BA	80 4F C7 52 58	0.811635157	Startwert für RND

Die Interpreterroutinen

Im Gegensatz zu den Integerzahlen stellt uns der Interpreter viele Routinen zur Verfügung, mit denen Rechenoperationen bei Fließkommazahlen durchgeführt werden können. Das Prinzip des Vorgehens ist dabei immer das gleiche: Zunächst wird der FAC mit der zu bearbeitenden Zahl geladen, dann wird die Rechenoperation durchgeführt. Das Ergebnis kann man schließlich wieder aus dem FAC ins RAM transferieren.

Wird auch der FAC II (ARG) benötigt, darf dieser erst nach dem FAC geladen werden, da zum Laden des FAC auch der FAC II benötigt wird. Hätte man also diesen zuerst geladen, wäre der darin befindliche Wert durch das Laden des FAC zerstört worden. Die folgenden Beispiele sollen die Verwendung der Rechenroutinen in Maschinensprache verdeutlichen. Als Operanden werden Konstanten des Betriebssystems benutzt. Dadurch brauchen wir vor der Rechnung nicht noch extra Fließkommazahlen im RAM anzulegen.

Arithmetikberechnungen mit dem FAC

a) FAC = FAC / 10 ($BAFE)

Wir wollen die Konstante 0.25 (ab $E2EA im ROM) durch 10 teilen. Dafür laden wir zunächst den FAC, wie wir es auch schon oben gemacht haben. Dann rufen wir die Rechenroutine auf, die den Wert durch 10 teilt. Anschließend wollen wir das Ergebnis in eine Variable speichern, deren Startadresse in $5F/$60 gespeichert ist. Zu dieser müssen wir noch zwei addieren, da ja der Variablenname übersprungen werden muß:

```
LDA #$EA      ;Lowbyte Startadresse
LDY #$E2      ;Highbyte derselben
JSR $BBA2     ;0.25 nach FAC laden
```

```
   JSR $BAFE     ;FAC=FAC/10 ausfuehren
   LDY $60       ;Highbyte Variablenadresse
   LDA $5F       ;Lowbyte Variablenadresse
   CLC           ;plus 2
   ADC #02
   BCC L1        ;kein Ueberlauf
   INY           ;Highbyte erhoehen
L1 TAX
   JSR $BBD4     ;FAC in Variable abspeichern
```

Die Speicherroutine wandelt die Fließkommazahl automatisch vom 6-Byte- ins 5-Byte-Format mit Vorzeichenbit um.

b) FAC = FAC * 10 ($BAE2)

Nun soll der Wert 0.707106781 (ab $B9D6) mit 10 multipliziert werden. Das Vorgehen ist das gleiche wie im vorigen Beispiel, der Unterschied besteht nur darin, daß jetzt die Multiplikations- statt die Divisionsroutine aufgerufen wird.

```
   LDA #$D6      ;Lowbyte Startadresse
   LDY #$B9      ;Highbyte derselben
   JSR $BBA2     ;0.70710... nach FAC laden
   JSR $BAE2     ;FAC=FAC*10 ausfuehren
   LDY $60       ;Highbyte Variablenadresse
   LDA $5F       ;Lowbyte Variablenadresse
   CLC           ;plus 2
   ADC #02
   BCC L1        ;kein Ueberlauf
   INY           ;Highbyte erhoehen
L1 TAX
   JSR $BBD4     ;FAC in Variable abspeichern
```

c) FAC = FAC + 0.5 ($B849)

Diese Routine wird zum Runden auf ganze Zahlen benötigt. In Basic sähe dies z.B. so aus: Zahl = INT(Zahl+0.5). Wir wollen zu der Konstanten 81.6052237 (ab $E2FF) die 0.5 addieren. Dazu gehen wir wie oben vor, rufen jedoch die Additionsroutine auf:

```
   LDA #$FF     ;Lowbyte Startadresse
   LDY #$E2     ;Highbyte derselben
   JSR $BBA2    ;81.6052237 nach FAC laden
   JSR $B849    ;FAC=FAC+0.5 ausfuehren
   LDY $60      ;Highbyte Variablenadresse
   LDA $5F      ;Lowbyte Variablenadresse
   CLC          ;plus 2
   ADC #02
   BCC L1       ;kein Ueberlauf
   INY          ;Highbyte erhoehen
L1 TAX
   JSR $BBD4    ;FAC in Variable abspeichern
```

d) FAC = Konstante / FAC ($BB0F)

Hierbei handelt es sich um eine Routine, die eine Konstante durch den FAC teilt. Vor dem Aufruf muß der Akku mit dem Lowbyte der Startadresse dieser Konstanten geladen werden, das Y-Register mit dem Highbyte. Sonst entspricht der Ablauf den vorhergehenden Beispielen. Wir wollen die Division der beiden Konstanten –0.5 (ab $B9E0) und 1.41421356 (ab $B9DB) durchführen.

```
   LDA #$E0    ;Lowbyte Startadresse 1.Konstante
   LDY #$B9    ;Highbyte derselben
   JSR $BBA2   ;-0.5 nach FAC laden
   LDA #$DB    ;Lowbyte Startadresse 2.Konstante
   LDY #$B9    ;Highbyte derselben
   JSR $BB0F   ;FAC=Konstante/FAC ausfuehren
   LDY $60     ;Highbyte Variablenadresse
   LDA $5F     ;Lowbyte Variablenadresse
   CLC         ;plus 2
   ADC #02
   BCC L1      ;kein Ueberlauf
   INY         ;Highbyte erhoehen
L1 TAX
   JSR $BBD4   ;FAC in Variable abspeichern
```

Mit dieser Routine ist es u.a. auch möglich, auf einfachste Weise den Kehrwert einer Zahl zu berechnen: Man braucht nur als Konstante eine 1 einzusetzen. Damit ergibt sich die Rechenvorschrift zu FAC = 1 / FAC.

e) FAC = Konstante FAC ($B850)

Hierbei wird der Inhalt des FAC von einer Konstanten subtrahiert. Die Vorgehensweise entspricht der des letzten Beispiels, die Konstanten wollen wir der Einfachheit halber unverändert übernehmen.

```
   LDA #$E0    ;Lowbyte Startadresse 1.Konstante
   LDY #$B9    ;Highbyte derselben
   JSR $BBA2   ;-0.5 nach FAC laden
   LDA #$DB    ;Lowbyte Startadresse 2.Konstante
   LDY #$B9    ;Highbyte derselben
   JSR $B850   ;FAC=Konstante-FAC ausfuehren
   LDY $60     ;Highbyte Variablenadresse
   LDA $5F     ;Lowbyte Variablenadresse
   CLC         ;plus 2
   ADC #02
   BCC L1      ;kein Ueberlauf
   INY         ;Highbyte erhoehen
L1 TAX
   JSR $BBD4   ;FAC in Variable abspeichern
```

f) FAC = Konstante + FAC ($B867)

Diese Routine stellt genau das Gegenteil zur letzten dar. Der Ablauf ist völlig identisch, nur daß hier die Konstante zum FAC addiert wird, statt subtrahiert. Als Konstanten wollen wir bei diesem Beispiel 999999999 (ab $BDB8) und 99999999.9 (ab $BDB3) verwenden.

```
    LDA #$B8     ;Lowbyte Startadresse 1.Konstante
    LDY #$BD     ;Highbyte derselben
    JSR $BBA2    ;999999999 nach FAC laden
    LDA #$B3     ;Lowbyte Startadresse 2.Konstante
    LDY #$BD     ;Highbyte derselben
    JSR $BB0F    ;FAC=Konstante+FAC ausfuehren
    LDY $60      ;Highbyte Variablenadresse
    LDA $5F      ;Lowbyte Variablenadresse
    CLC          ;plus 2
    ADC #02
    BCC L1       ;kein Ueberlauf
    INY          ;Highbyte erhoehen
L1  TAX
    JSR $BBD4    ;FAC in Variable abspeichern
```

g) FAC = Konstante * FAC ($BA28)

Hierbei wird der FAC mit einer Konstanten multipliziert. Die Konstanten wollen wir vom letzten Beispiel übernehmen, wodurch eine riesige Zahl entsteht (9.99 * 10^16).

```
    LDA #$B8     ;Lowbyte Startadresse 1.Konstante
    LDY #$BD     ;Highbyte derselben
    JSR $BBA2    ;999999999 nach FAC laden
    LDA #$B3     ;Lowbyte Startadresse 2.Konstante
    LDY #$BD     ;Highbyte derselben
    JSR $BA28    ;FAC=Konstante*FAC ausfuehren
    LDY $60      ;Highbyte Variablenadresse
    LDA $5F      ;Lowbyte Variablenadresse
    CLC          ;plus 2
    ADC #02
    BCC L1       ;kein Ueberlauf
    INY          ;Highbyte erhoehen
L1  TAX
    JSR $BBD4    ;FAC in Variable abspeichern
```

h) FAC = NOT FAC ($AED4)

Diese interessante Routine funktioniert so: Da man im Fließkommaformat ja keine Bits umdrehen kann, wird der FAC zunächst nach Integer gewandelt. Anschließend werden alle Bits dieser Integerzahl umgedreht und das Ergebnis wieder nach Fließkomma gewandelt. Wir wollen hier einmal 2*PI (ab $E2E5) invertieren:

```
    LDA #$E5     ;Lowbyte Startadresse
    LDY #$E2     ;Highbyte derselben
```

```
   JSR $BBA2    ;2*PI nach FAC laden
   JSR $AED4    ;FAC=NOT FAC ausfuehren
   LDY $60      ;Highbyte Variablenadresse
   LDA $5F      ;Lowbyte Variablenadresse
   CLC          ;plus 2
   ADC #02
   BCC L1       ;kein Ueberlauf
   INY          ;Highbyte erhoehen
L1 TAX
   JSR $BBD4    ;FAC in Variable abspeichern
```

i) FAC = FAC ($BFB4)

Bei der Besprechung der Integerzahlen mußten wir uns eine Routine zum Umdrehen des Vorzeichens selbst schreiben, da sie vom Interpreter nicht zur Verfügung gestellt wird. Bei der Fließkommarechnung existiert eine Routine zu diesem Zweck bereits. Wir wollen aus 2*PI einfach einmal –2*PI machen:

```
   LDA #$E5     ;Lowbyte Startadresse
   LDY #$E2     ;Highbyte derselben
   JSR $BBA2    ;2*PI nach FAC laden
   JSR $BFB4    ;FAC=-FAC ausfuehren
   LDY $60      ;Highbyte Variablenadresse
   LDA $5F      ;Lowbyte Variablenadresse
   CLC          ;plus 2
   ADC #02
   BCC L1       ;kein Ueberlauf
   INY          ;Highbyte erhoehen
L1 TAX
   JSR $BBD4    ;FAC in Variable abspeichern
```

j) FAC mit Konstanten vergleichen ($BC5B)

Zwei Fließkommazahlen »per Hand« zu vergleichen, ist sehr umständlich. Auch für diesen Vorgang existiert daher bereits eine ROM-Routine. Der Ablauf des Programms ist identisch mit dem der Rechenroutinen wie z.B. FAC=Konstante-FAC. Als Ergebnis des Vergleichs erhält man einen Wert im Akku, den man wie folgt interpretieren muß:

FAC größer Konstante: Akku=1
FAC gleich Konstante: Akku=0
FAC kleiner Konstante: Akku=255

Um dieses Ergebnis möglichst einfach auswerten zu können, geht man am besten so vor: Zunächst prüft man mit dem BEQ-Befehl, ob eine Gleichheit vorliegt. Die Unterscheidung zwischen 1 und 255 kann man dadurch treffen, daß man das 7. Bit mittels des BPL- oder BMI-Befehls prüft. Eine weitere Möglichkeit liegt darin, zunächst einen ROL-Befehl durchzuführen und dann mit dem BCC-Befehl das Carry-Flag zu überprüfen, da sich ja das 7. Bit nach

dem ROL-Befehl in diesem befindet. Wir wollen PI (ab $AEA8) mit PI/2 (ab $E2E0) vergleichen, was natürlich ein »größer« als Ergebnis zur Folge hat.

```
LDA #$A8     ;Lowbyte Startadresse 1.Konstante
LDY #$AE     ;Highbyte derselben
JSR $BBA2    ;PI nach FAC laden
LDA #$E0     ;Lowbyte Startadresse 2.Konstante
LDY #$E2     ;Highbyte derselben
JSR $BC5B    ;FAC mit PI/2 vergleichen
BEQ GLEICH   ;FAC=Konstante
BPL GRÖSSER  ;FAC groesser als Konstante
KLEINER. ...;Maschinenroutine nach kleiner
.............
GLEICH......;Maschinenroutine nach gleich
.............
GRÖSSER.....;Maschinenroutine nach groesser
............(wird hier angesprungen)
```

k) Vorzeichen des FAC prüfen ($BC2B)

Eine Vorzeichenprüfung benötigt man immer dann, wenn das Ergebnis nicht eindeutig ist, wie z.B. bei den Winkelfunktionen. Wenn man beispielsweise von einem Winkel den Sinus- sowie den Cosinuswert besitzt, ist dieser Winkel erst dann eindeutig festgelegt, wenn man die Vorzeichen der beiden Winkelfunktionen kennt, z.B.

```
SIN(45 Grad) = +0.707  ;  COS(45 Grad) = +0.707
SIN(135 Grad)= +0.707  ;  COS(135 Grad)= -0.707
```

Ohne Vorzeichenprüfung könnte man diese beiden Winkel nicht unterscheiden. Die ROM-Routine liefert das Ergebnis wiederum im Akku ab:

negative Zahl: Akku = 255
Zahl = 0: Akku = 0
positive Zahl: Akku = 1

Wir wollen das Vorzeichen der Konstanten +0.25 (ab $E2EA) prüfen:

```
LDA #$EA     ;Lowbyte Startadresse
LDY #$E2     ;Highbyte derselben
JSR $BBA2    ;0.25 nach FAC laden
JSR $BC2B    ;Vorzeichen pruefen
BEQ GLEICH   ;FAC gleich Null
BPL GRÖSSER  ;FAC groesser Null
KLEINER. ...;Maschinenroutine nach kleiner
.............
GLEICH......;Maschinenroutine nach gleich
.............
GRÖSSER.....;Maschinenroutine nach groesser
............(wird hier angesprungen)
```

l) Polynomberechnung I ($E059)

Das größte Problem für einen Computer stellen die transzendenten Funktionen wie SIN, COS, LOG usw. dar. Diese können nämlich nicht exakt durch die vier Grundrechenarten berechnet werden, sondern müssen durch ein Näherungsverfahren bestimmt werden, das natürlich keine exakten Werte liefern kann. Beim C64 werden die meisten dieser Funktionen durch sogenannte Polynome angenähert, das sind Funktionen der Form

$$y = a0 + a1*x + a2*x\uparrow 2 + a3*x\uparrow 3 + a4*x\uparrow 4 + a5*x\uparrow 5 + a6*x\uparrow 6 + \ldots$$

Das Ergebnis wird um so genauer, je mehr Glieder ein Polynom aufweist, natürlich erhöht sich dabei aber auch die Rechenzeit, so daß man immer einen Kompromiß zwischen Rechenzeit und Genauigkeit finden muß.

Zur Berechnung eines Polynoms n-ten Grades (d.h. die höchste vorkommende Potenz von »X« ist »n«) sind allgemein n*(n+1)/2 Multiplikationen und »n« Additionen erforderlich, wie man sich sofort z.B. bei einem Polynom 4. Grades klarmachen kann:

n=4: Es sind 1+2+3+4=10 Multiplikationen und 4 Additionen nötig.

Es gibt jedoch ein Verfahren, das die Polynomberechnung wesentlich vereinfacht. Dazu wird die obige Gleichung umgewandelt:

$$y = (((((a6*x+a5)*x+a4)*x+a3)*x+a2)*x+a1)*x+a0$$

Hierbei sind offenbar nur noch 6 Multiplikationen und 6 Additionen erforderlich, allgemein gesprochen jeweils 2*n Rechenoperationen. Dieses Verfahren benutzt auch der Basic-Interpreter zur Berechnung der transzendenten Funktionen. Wenn wir diese Routine benutzen wollen, müssen wir eine Tabelle anlegen, aus der der Grad des Polynoms als 1-Byte-Wert sowie die Koeffizienten »a0« bis »an« im 5-Byte-Fließkommaformat hervorgehen. Dabei muß folgende Reihenfolge eingehalten werden:

1:	Polynomgrad n
2:	Koeffizient a(n)
3:	Koeffizeint a(n–1)
4:	Koeffizient a(n–2)
n–1:	Koeffizient a2
n:	Koeffizient a1
n+1:	Koeffizient a0

Die Anzahl der Koeffizienten ist immer um eins höher als der Grad des Polynoms, da das letzte Glied »a0« ja variablenfrei ist. Theoretisch könnte man Polynome bis zum Grad 255 verwenden, wenn man den 1-Byte-Wert voll ausnutzt. Bevor wir weitermachen, wollen wir erst mal eine solche Tabelle einrichten. Wir bedienen uns des einfachen Beispiels:

$$y = 0.25 * x\uparrow 3 \quad 0.5 * x\uparrow 2 + 1 * x\uparrow 1 + 0.25$$

Der Grad des Polynoms ist also 3, die Koeffizienten in absteigender Reihenfolge lauten:

a3 = 0.25
a2 = –0.5
a1 = 1
a0 = 0.25

Diese wurden bewußt so gewählt, da diese Zahlen komplett im ROM im Fließkommaformat abgespeichert sind. Unser Tabellenaufbau soll ab $033C (Start des Kassettenpuffers) beginnen. Zunächst müssen wir den Exponenten ablegen und dann die 4 Koeffizienten im Abstand von jeweils 5 Byte. Dazu holen wir sie zunächst in den FAC und speichern sie dann in das RAM ab, wie wir das bislang auch schon gemacht haben.

```
LDA #03             ;Exponent
STA $033C           ;abspeichern
LDA #$EA            ;0.25 aus dem ROM
LDY #$E2            ;ab $E2EA
JSR $BBA2           ;in den FAC holen
LDX #$3D            ;als a3 ab $033D
LDY #$03            ;ablegen
JSR $BBD4
LDX #$3D+15         ;und als a0 ab $033D+15
LDY #$03            ;ablegen
JSR $BBD4
LDA #$E0            ;-0.5 aus dem ROM
LDY #$B9            ;ab $B9E0
JSR $BBA2           ;in den FAC holen
LDX #$3D+5          ;als a2 ab $033D+5
LDY #$03            ;ablegen
JSR $BBD4
LDA #$BC            ;1 aus dem ROM
LDY #$B9            ;ab $B9BC
JSR $BBA2           ;in den FAC holen
LDX #$3D+10         ;als a1 ab $033D+10
LDY #$03            ;ablegen
JSR $BBD4
```

Nach diesen umfangreichen Vorbereitungen können wir nun zur eigentlichen Polynomberechnung kommen. Dafür muß das Argument (hier: x) in den FAC gebracht und anschließend der Akku (Lowbyte) und das Y-Register (Highbyte) mit der Startadresse der Tabelle geladen werden. Dann kann endlich die Polynomberechnung aufgerufen werden. Das Ergebnis wollen wir wieder in unsere Variable, deren Startadresse ja in $5F/$60 steht, abspeichern. Als X-Wert nehmen wir PI, so daß wir ihn aus dem ROM holen können. Damit rechnen wir also folgendes:

0.25*(PI↑3)-0.5*(PI↑2)+1*(PI)+0.25 = ???

```
LDA #$A8            ;PI aus dem ROM
LDY #$AE            ;ab $AEA8
```

```
    JSR $BBA2          ;in den FAC holen
    LDA #$3C           ;Beginn der Tabelle (Lowbyte)
    LDY #$03           ;Beginn der Tabelle (Highbyte)
    JSR $E059          ;Polynom berechnen
    LDY $60            ;Startadresse Variable (Highbyte)
    LDA $5F            ;Startadresse Variable (Lowbyte)
    CLC
    ADC #02            ;plus 2
    BCC L1             ;kein Ueberlauf
    INY
L1  TAX
    JSR $BBD4          ;Ergebnis in Variable ablegen
```

In unserem Fall mit x=PI=3.14159265 kommt als Ergebnis 6,2083 heraus.

m) Polynomberechnung II ($E043)

Der Interpreter des C64 enthält neben dieser universellen noch eine zweite spezielle Routine zur Polynomberechnung. Diese rechtfertigt ihre Existenz dadurch, daß bei vielen transzendenten Funktionen das Näherungspolynom diese Form annimmt. Dabei werden nur Terme mit ungeradem Exponenten für »x« zugelassen. Das Polynom hat daher die Form

y = a0 * x + a1 * x↑3 + a2 * x↑5 + ...

Dieses Polynom erhält man aus dem vorher besprochenen dadurch, daß man das Argument »x« durch (x^2) ersetzt und den so entstandenen Term nochmals mit »x« multipliziert.

y = x * (a0 + a1 * (x↑2) + a2 * ((x↑2)↑2) + a3 * ((x↑2)↑3) +...

Der Berechnungsvorgang verläuft analog zur vorigen Polynomberechnung. Wir wollen die Gleichung

10 * x + 1 * x↑3 + (2*PI) * x↑5 = ??? lösen.

Als Polynomgrad ergibt sich 2 (nicht 5, da wir ja als Argument »x↑2« verwenden und anschließend nochmals mit »x« multiplizieren), die Koeffizienten lauten:

a2 = PI*2
a1 = 1
a0 = 10

Als erstes legen wir also wieder eine Tabelle am Anfang des Kassettenpuffers an.

```
LDA #02         ;Exponent
STA $033C       ;abspeichern
LDA #$E5        ;PI*2 aus dem ROM
LDY #$E2        ;ab $E2E5
JSR $BBA2       ;in den FAC holen
LDX #$3D        ;als a2 ab $033D
LDY #$03        ;ablegen
```

```
JSR $BBD4
LDA #$E8            ;1 aus dem ROM
LDY #$BF            ;ab $BFE8
JSR $BBA2           ;in den FAC holen
LDX #$3D+5          ;als a1 ab $033D+5
LDY #$03            ;ablegen
JSR $BBD4
LDA #$F9            ;10 aus dem ROM
LDY #$BA            ;ab $BAF9
JSR $BBA2           ;in den FAC holen
LDX #$3D+10         ;als a0 ab $033D+10
LDY #$03            ;ablegen
JSR $BBD4
```

Nun wollen wir das Polynom mit dem Argument x = 0.19999912 berechnen und das Ergebnis wieder in unsere Variable übertragen:

```
   LDA #$6C         ;0.19999912 aus dem ROM
   LDY #$E3         ;ab $E36C
   JSR $BBA2        ;in den FAC holen
   LDA #$3C         ;Beginn der Tabelle (Lowbyte)
   LDY #$03         ;Beginn der Tabelle (Highbyte)
   JSR $E043        ;Polynom berechnen
   LDY $60          ;Startadresse Variable (Highbyte)
   LDA $5F          ;Startadresse Variable (Lowbyte)
   CLC
   ADC #02          ;plus 2
   BCC L1           ;kein Ueberlauf
   INY
L1 TAX
   JSR $BBD4        ;Ergebnis in Variable ablegen
```

Als Ergebnis erhalten wir mit x = 0.19999912

10 * x + 1 * x↑3 + (2*PI) * x↑5 = 2.01

Funktionsberechnung mit dem FAC

Der Basic-Interpreter des C64 stellt dem Benutzer eine Fülle von Funktionen zur Verfügung, die teilweise jedoch wie oben beschrieben nur näherungsweise berechnet werden können und auch relativ viel Zeit zur Ausführung benötigen. Die Berechnung erfolgt analog zur Arithmetik, d.h. zunächst wird der FAC mit dem Argument geladen. Nach der Rechenoperation steht der Funktionswert im FAC zur Verfügung. Als Funktionsargument wollen wir nun keine Konstante mehr nehmen, sondern den Wert einer Variablen, deren Startadresse in $5F/$60 gespeichert ist. Anschließend soll der Funktionswert wieder in diese Variable übertragen werden. In Basic könnte man dies so formulieren, wenn die Variable A hieße und die Cosinus-Funktion benützt würde:

A = COS (A)

a) FAC = ABS(FAC) ($BC58)

Durch die ABS-Funktion wird der Betrag einer Zahl erzeugt, d.h. falls diese positiv ist, bleibt sie unverändert, bei einer negativen Zahl wird das Vorzeichen gewechselt.

```
    LDY $60       ;Start der Variablen (Highbyte)
    LDA $5F       ;Start der Variablen (Lowbyte)
    CLC
    ADC #02       ;plus 2
    BCC L1        ;kein Ueberlauf
    INY           ;Highbyte plus 1
L1  STA $FA       ;Startadresse
    STY $FB       ;merken
    JSR $BBA2     ;FAC mit Variablen-Wert laden
L2  JSR $BC58     ;ABS-Funktion ausfuehren
    LDY $FB       ;Startadresse der Variablen
    LDX $FA       ;wiederholen
    JSR $BBD4     ;Funktionswert uebertragen
```

Dieses Verfahren ist für viele Funktionen identisch. Man braucht nur in der Zeile L2 die Funktionsadresse der jeweiligen Funktion einzusetzen. Ich möchte daher die Funktionen mit gleichem Rechengang nur kurz ohne Beispielprogramm aufführen:

b) FAC = ATN(FAC) (Arcustangens-Funktion ab $E30E)
c) FAC = COS(FAC) (Cosinus-Funktion ab $E264)
d) FAC = EXP(FAC) (Potenz zur Basis e ab $BFED)
e) FAC = FRE(FAC) (Freier Speicherplatz ab $B37D)
f) FAC = INT(FAC) (Ganzzahliger Anteil ab $BCCC)
g) FAC = LOG(FAC) (Natürlicher Logarithmus ab $B9EA)
h) FAC = POS(FAC) (Cursorspalte ab $B39E)
i) FAC = RND(FAC) (Zufallszahl ab $E097)
j) FAC = SGN(FAC) (Vorzeichen ab $BC39)
k) FAC = SIN(FAC) (Sinus-Funktion ab $E26B)
l) FAC = SQR(FAC) (Quadratwurzel ab $BF71)
m) FAC = TAN(FAC) (Tangens-Funktion ab $E2B4)

Die Funktionsargumente der transzendenten Funktionen wie Sinus, Cosinus usw. müssen im Bogenmaß in den FAC geladen werden. Zur Umrechnung gilt:

2*PI = 360 Grad

Sehr oft liegt jedoch ein Winkel im Gradmaß vor, so daß er vor dem Beginn der Rechenoperation erst in das Bogenmaß umgewandelt werden muß. Anschließend kann man das Ergebnis wieder ins Gradmaß zurückrechnen. Für die Umrechnung vom Grad- ins Bogenmaß gilt folgende Formel:

Winkel(Bogenmaß) = (Winkel(Gradmaß)/360)*2*PI

Der umgedrehte Weg sieht so aus:

Winkel(Gradmaß) = (Winkel(Bogenmaß)/(2*PI))*360

Wir wollen als Beispiel den Sinus des Winkels 120 Grad berechnen. Dazu wandeln wir ihn erst ins Bogenmaß um und rufen dann die Sinus-Routine auf. Als erstes müssen wir den Wert 120 im RAM anlegen. Dazu laden wir ihn mit der Ein-Byte-Routine ohne Vorzeichen und speichern ihn an den Anfang des Kassettenpuffers ($033C) ab. Anschließend laden wir den FAC mit der Zahl 360, indem wir die Zwei-Byte-Routine ohne Vorzeichen benutzen. Dann kann die Division 120/360 durchgeführt werden:

```
LDY #120   ;120 in
JSR $BC3C  ;FAC einladen
LDX #$3C   ;FAC nach
LDY #$03   ;$033C
JSR $BBD4  ;abspeichern
LDA #01    ;Highbyte von 360
LDY #104   ;Lowbyte von 360
STA $62
STY $63
SEC
LDX #144
JSR $BC49  ;360 in FAC laden
LDA #$3C   ;Zeiger auf 120
LDY #$03   ;ab $033C
JSR $BB0F  ;Division ausfuehren
```

Als nächstes müssen wir das Ergebnis mit 2*PI multiplizieren. Da 2*PI als Konstante bereits im ROM steht (ab $E2E5), können wir den FAC sofort mit ihr multiplizieren und anschließend den Sinus berechnen:

```
LDA #$E5   ;Zeiger auf 2*PI
LDY #$E2   ;ab $E2E5
JSR $BA28  ;Multiplikation ausfuehren
JSR $E26B  ;Sinus berechnen
```

Hier die benötigten Rechenzeiten der einzelnen Funktionen in aufsteigender Reihenfolge. Als Funktionsargument wurde jeweils PI verwendet:

Name	Rechenzeit (ms)
ABS	0.0
POS	0.3
SGN	0.4
FRE	0.6
INT	0.9
RND	3.5

Name	Rechenzeit (ms)
RND	3.5
LOG	22.2
SIN	24.5
EXP	26.6
COS	27.9
ATN	44.6
TAN	49.8
SQR	51.2

Arithmetikberechnungen mit FAC und ARG

Die Vorgehensweise bei der Verknüpfung beider Fließkomma-Akkus ähnelt sehr der des letzten Abschnitts. Zu beachten ist, wie schon erwähnt, nur, daß der FAC immer vor dem ARG geladen werden muß. Außerdem ist es bei einigen Verknüpfungen erforderlich, vor dem Aufruf der Rechenroutine den Akku mit dem Exponenten des FAC zu laden, was man einfach durch den Befehl

```
LDA $61
```

erreichen kann.

a) FAC = ARG / FAC ($BB12)

Unser Beispiel zur Divisionsroutine soll die ROM-Konstante PI (ab $E2E0) durch PI/2 (ab $AEA8) teilen. Dafür müssen diese erst einmal in FAC und ARG geladen werden. Vor dem Aufruf der Routine muß der Akku mit dem Exponenten des FAC geladen werden.

```
LDA #$E0    ;Startadresse PI
LDY #$E2    ;($E2E0)
JSR $BBA2   ;FAC mit PI laden
LDA #$A8    ;Startadresse PI/2
LDY #$AE    ;($AEA8)
JSR $BA8C   ;ARG mit PI/2 laden
LDA $61     ;Akku mit Exponenten laden
JSR $BB12   ;Division ausfuehren
```

b) FAC = ARG + FAC ($B86A)

Wir wollen die Zahlen 10 und 20 addieren. Während 10 als Konstante im ROM vorhanden ist (ab $BAF9), müssen wir 20 »per Hand« einladen. Da es sich hierbei um einen positiven Ein-Byte-Wert handelt, können wir die entsprechende Routine aus Kapitel 3.1.4.1. benutzen. Der Akku muß wiederum mit dem Exponenten des FAC geladen werden.

```
LDY #20     ;20 in
JSR $B3A2   ;FAC einladen
```

```
LDA #$F9     ;Startadresse 10
LDY #$BA     ;($BAF9)
JSR $BA8C    ;ARG mit 10 laden
LDA $61      ;Akku mit Exponenten laden
JSR $B86A    ;Addition ausfuehren
```

c) FAC = ARG FAC ($B853)

In diesem Beispiel soll der Wert einer Variablen, deren Startadresse in $5F/$60 zu finden ist, von der ROM-Konstanten 1 (ab $E376) subtrahiert werden. Vor dem Aufruf der Routine muß der Akku erneut mit dem Exponenten des FAC versorgt werden.

```
   LDY $60   ;Variablenstart (Highbyte)
   LDA $5F   ;Variablenstart (Lowbyte)
   CLC
   ADC #02   ;plus 2
   BCC L1    ;kein Ueberlauf
   INY       ;Highbyte erhoehen
L1 JSR $BBA2 ;FAC mit Variablenwert laden
   LDA #$76  ;Startadresse von 1
   LDY #$E3  ;($ E376)
   JSR $BA8C ;ARG mit 1 laden
   LDA $61   ;Akku mit Exponenten laden
   JSR $B853 ;Subtraktion ausfuehren
```

d) FAC = ARG * FAC ($BA2B)

Wir möchten die Zahl 300 mit sich selbst multiplizieren. Dafür laden wir sie zunächst in den FAC und transferieren sie von dort aus in den ARG. Da 300 nicht als ROM-Konstante vorliegt, müssen wir sie durch die 2-Byte-Wert-Routine ohne Vorzeichenberücksichtigung aus Kapitel 3.1.4.1. in den FAC einladen. Der Akku muß wiederum den Exponenten enthalten, bevor die Routine aufgerufen wird.

```
LDY #44      ;Lowbyte von 300
LDA #01      ;Highbyte von 300
STA $62
STY $63
SEC
LDX #144
JSR $BC49    ;300 in FAC einladen
JSR $BC0C    ;FAC in ARG uebertragen
LDA $61      ;Exponent in Akku laden
JSR $BA2B    ;Multiplikation ausfuehren
```

e) FAC = ARG ↑ FAC ($BF7B)

Wir möchten die Rechnung PI↑4 ausführen. Da vier nicht als ROM-Konstante vorliegt, könnten wir diese Zahl wieder per Hand einladen. Ich möchte jedoch einen anderen, wenn auch umständlicheren Weg gehen: Da der Kehrwert von 4 = $^1/_4$ = 0.25 im ROM gespeichert

ist (ab $E2EA), bilden wir zunächst den Kehrwert von 4 und laden diesen dann in den FAC ein. Zum letzten Mal muß der Akku mit dem Exponenten geladen werden.

```
LDA #$EA    ;Startadresse von 0.25
LDY #$E2    ;($E2EA)
JSR $BBA2   ;0.25 in FAC laden
LDA #$BC    ;Startadresse von 1
LDY #$B9    ;($B9BC)
JSR $BB0F   ;1/0.25 rechnen
LDA #$A8    ;Startadresse von PI
LDY #$AE    ;($AEA8)
JSR $BA8C   ;PI in ARG einladen
LDA $61     ;FAC-Exponent in Akku laden
JSR $BF7B   ;PIì4 ausrechnen
```

f) FAC = ARG ↑ Konstante ($BF78)

In diesem Beispiel wollen wir die Zahl 200 mit dem Wert SQR(2) potenzieren. Dafür müssen wir die Zahl »per Hand« in den FAC laden und von dort in den ARG transferieren. Der Wert SQR(2) liegt als Konstante ab der Adresse $B9DB. Vor dem Aufruf der Routine müssen Lowbyte im Akku und Highbyte im Y-Register übergeben werden. Für das Einladen der Zahl 200 nehmen wir wieder die Ein-Byte-Routine aus Kapitel 3.1.4.1.

```
LDA #200    ;200 in FAC
JSR $B3A2   ;einladen
JSR $BC0C   ;FAC in ARG uebertragen
LDA #$DB    ;Startadresse von SQR(2)
LDY #$B9    ;($B9DB)
JSR $BF78   ;Potenzierung ausfuehren
```

g) FAC = ARG OR FAC ($AFE6)

h) FAC = ARG AND FAC ($AFE9)

Die letzten beiden Routinen benötigen keinerlei Parameter beim Aufruf, deshalb habe ich auf die Beispiele verzichtet. Man lädt einfach FAC und ARG mit den entsprechenden Werten und kann die Routinen sofort aufrufen.

3.1.4.2 Übersicht über alle Fließkommaroutinen

Hier nun alle Routinen in chronologischer Reihenfolge:

Adresse	Zeiger auf Konstante	Akku	Funktion
$AED4			FAC = NOT FAC
$AFE6			FAC = ARG OR FAC
$AFE9			FAC = ARG AND FAC
$B849			FAC = FAC + 0.5

Adresse	Zeiger auf Konstante	Akku	Funktion
$B850	A/Y		FAC = Konstante/FAC
$B853		$61	FAC = ARG FAC
$B867	A/Y		FAC = Konstante+FAC
$B86A		$61	FAC = ARG + FAC
$BA28	A/Y		FAC = Konstante*FAC
$BA2B		$61	FAC = ARG * FAC
$BA8C	A/Y		ARG = Konstante
$BAE2			FAC = FAC * 10
$BAFE			FAC = FAC / 10
$BB0F	A/Y		FAC = Konstante/FAC
$BB12		$61	FAC = ARG / FAC
$BBA2	A/Y		FAC = Konstante
$BBD4	X/Y		Konstante = FAC
$BBFC			FAC = ARG
$BC0C			ARG = FAC
$BC1B			FAC runden
$BC5B	A/Y		FAC CMP Konstante
$BF78	A/Y		FAC = ARG↑Konstante
$BF7B		$61	FAC = ARG ↑ FAC
$BFB4			FAC = FAC
$E043	A/Y		FAC = Polynom II
$E059	A/Y		FAC = Ploynom I

3.1.4.3 Bildschirmausgabe einer Fließkommazahl

Eine Fließkommazahl wird durch die Interpreter-Routine

```
JSR $AABC
```

ausgegeben. Hierzu muß die Zahl im FAC stehen, der durch die Ausführung der Routine verändert wird. Man kann daher nach allen Berechnungsroutinen die Ausgabe sofort aufrufen, da das Ergebnis ja immer im FAC steht.

Falls das Ergebnis nach der Ausgabe nochmals benötigt wird, muß man es unbedingt im RAM zwischenspeichern.

3.1.5 Umwandlung der Variablenformate

Nachdem wir nun alle Variablentypen ausführlich besprochen haben, kommen wir zu einer interessanten Möglichkeit des Basic-Interpreters: der Umwandlung eines Variablentyps in einen anderen. Insgesamt stehen vier Möglichkeiten zur Verfügung:

a) Integerzahl in Fließkommazahl

Nach Übergabe der Integerzahl in Akku (Highbyte) und Y-Register (Lowbyte) kann man die Routine mit

```
JSR $B391
```

aufrufen. Die erzeugte Fließkommazahl wird im FAC abgelegt und kann dort weiterverarbeitet werden.

b) Fließkommazahl in Integerzahl

Die Fließkommazahl wird im FAC erwartet. Mit

```
JSR $B7F7
```

wird diese in eine Integerzahl umgewandelt, die im Akku (Highbyte) und Y-Register (Lowbyte) zur Verfügung steht. Eventuelle Nachkommastellen werden natürlich abgeschnitten.

c) String in Fließkommazahl

Da man mit Strings nicht rechnen kann, auch wenn sie ausschließlich aus Zahlen bestehen, kann man auch sie ins Fließkomma-, nicht jedoch ins Integer-Format umwandeln. Dazu muß die Startadresse des Strings in den Zellen $22 und $23 abgelegt werden, die Länge des Strings kommt in den Akku. Der Fließkommawert steht dann im FAC zur Verfügung. Wir wollen einmal eine Stringvariable, deren Startadresse in $5F/$60 abgelegt sei, ins Fließkommaformat umwandeln. Die Umwandlungsroutine beginnt bei $B7B5.

```
LDY #02        ;Index für Stringpointer
LDA ($5F),Y    ;Stringlaenge holen
PHA            ;und merken
INY
LDA ($5F),Y    ;Startadresse String Lowbyte
STA $22        ;nach $22
INY
LDA ($5F),Y    ;Startadresse String Highbyte
STA $23        ;nach $23
PLA            ;Laenge wieder holen
JSR $B7B5      ;String in Zahl umwandeln
```

d) Fließkommazahl in String

Hierbei wird der FAC in einen String umgewandelt. Dieser wird ab der Adresse 256 ($0100) angelegt, welche auch im Akku (Lowbyte) und Y-Register (Highbyte) zurückgegeben wird. Als erstes Zeichen steht in jedem Fall das Vorzeichen, bei positiven Zahlen ein Leerzeichen. Die erste Ziffer steht also immer erst an der Position 257 ($0101). Die Routine wird mit

```
JSR $BDDD
```

aufgerufen. Hat die Zahl einen kleineren Wert als 0.01, wird der String in der Form Mantisse * 10 ↑ Exponent angelegt. Um dies zu überprüfen, muß man die Speicherstelle $5E aus-

lesen. Falls eine gewöhnliche Dezimalzahl erzeugt wurde, hat sie eine Null zum Inhalt. Sonst ist in ihr der Exponent zur Basis 10 gespeichert. Über die Länge des Strings ist grundsätzlich nichts bekannt. Da er aber mit einem Null-Byte abgeschlossen wird, kann man sein Ende natürlich leicht feststellen. Zur Ausgabe würde sich die Routine

```
JSR $AB1E
```

anbieten. Diese hat die Eigenschaft, daß man die Stringlänge nicht mit übergeben muß, sondern nur den Stringstart in Akku (Lowbyte) und Y-Register (Highbyte). Es werden so lange Zeichen ausgegeben, bis ein Null-Byte gefunden wird. So kann man auf einfachste Weise eine Zahl umwandeln und als String ausgeben. Wir wollen in unserem Beispiel die Zahl PI in einen String umwandeln und diesen anschließend auf dem Bildschirm ausgeben:

```
LDA #$A8      ;Startadresse von PI
LDY #$AE      ;($AEA8) im Fließkommaformat
JSR $BBA2     ;in FAC einladen
JSR $BDDD     ;in String umwandeln
JSR $AB1E     ;und ausgeben
```

Da nach dem Rücksprung von JSR $BDDD die Startadresse des Strings schon in Akku und Y-Register gespeichert sind, braucht man diese vor dem Aufruf der Ausgaberoutine nicht extra zu laden.

3.1.6 Einrichten/Suchen einer nichtindizierten Variablen

Nachdem wir nun alles über den Aufbau der Variablen kennen und auch wissen, wie man mit ihnen rechnen kann, stellt sich aber noch die Frage, wo und wie man diese im Speicher verwalten kann. Nun, die Lage im RAM ist prinzipiell beliebig. Um möglichst Arbeit zu sparen, sollte man zum Suchen und Einrichten einer Variablen die Interpreterroutinen benutzen. Diese erwarten die Start- und Endadresse des Variablenbereiches in jeweils zwei Speicherstellen:

Start der Variablen: $2D (Lowbyte) und $2E (Highbyte)
Ende der Variablen: $31 (Lowbyte) und $32 (Highbyte)

Nicht zulässig sind die RAM-Bereiche unter den ROMs, da die Interpreterroutinen den Prozessorport nicht auf RAM umschalten, bevor sie die Variableninhalte holen.

Ein Problem tritt bei den Strings auf: Wie wir wissen, wird in der Variablentabelle nur der sogenannte Descriptor gespeichert. Die eigentlichen Zeichenketten werden normalerweise vom Interpreter von $A000 abwärts angelegt, laufen also den Variablen entgegen, die sich in Basic vom Programmende aufwärts bewegen. Will man dieses Konzept aufrechterhalten, muß man unbedingt aufpassen, daß sich Variablen und Zeichenketten nicht überschneiden. Daher halte ich es immer für sinnvoller, diese Bereiche in Maschinensprache zu trennen. Die Adresse, von der die Strings abwärts aufgebaut werden, wird in den Adressen $37 (Lowbyte) und $38 (Highbyte) festgelegt.

Zum Suchen bzw. Einrichten einer nichtindizierten Variablen dient die Routine

```
JSR $B0E7
```

Dazu muß vor dem Aufruf der Variablenname in den Speicherstellen $45 (erstes Zeichen) und $46 (zweites Zeichen) gespeichert werden (jeweils der ASCII-Code). Falls der Name nur ein Zeichen enthält, muß die Adresse $46 mit einer Null gefüllt werden (nicht mit deren ASCII-Code !!!). Das Ergebnis der Suche ist ein Zeiger, der auf den Inhalt der Variablen zeigt, bei Integer- und Fließkommazahlen also direkt auf die Zahl, bei Strings auf den Descriptor. Diese Adresse wird in den Zellen $47/$48 und zusätzlich im Akku (Lowbyte) und Y-Register (Highbyte) übergeben. Falls die Suche erfolglos blieb, wird die angesprochene Variable neu hinter der letzten existierenden Variablen angelegt. Auch hier erhält man einen Zeiger auf den Variableninhalt, der bei Zahlen eine Null und bei Strings einen Leerstring (Länge Null) darstellt.

Jetzt kann man die Variablen benutzen, d.h. deren Werte verarbeiten oder ihnen neue zuweisen.

Wir wollen einmal den Inhalt der Fließkommavariablen »FR« in den FAC übertragen:

```
    LDA #"F"
    STA $45       ;erstes Zeichen Variablenname
    LDA #"R"
    STA $46       ;zweites Zeichen Variablenname
    (JSR INT)     ;nur bei Integer-Variablen ausfuehren
    (JSR STR)     ;nur bei Strings ausfuehren
    JSR $B0E7     ;Variablenadresse holen
    JSR $BBA2     ;Wert in FAC uebertragen
    ;
INT LDA #$80      ;Bit 7 des ersten
    ORA $45       ;und zweiten
    STA $45       ;Buchstabens
STR LDA #$80      ;des Variablennamens
    ORA $46       ;setzen
    STA $46
    RTS
```

Die Unterprogramme INT und STR haben die Aufgabe, je nach Variablentyp die Bits 7 der Codes der Namenszeichen zu setzen. Der direkte Einsprung nach $BBA2 ist deshalb möglich, da die Startadresse ja in Akku und Y-Register übergeben werden muß. Genau in diesem Format aber bekommt man sie ja von der Routine $B0E7 geliefert, so daß komplizierte Rechengänge entfallen.

3.1.7 Wertetabelle in Maschinensprache

Nach endlos langer Theorie möchte ich Ihnen hiermit ein Programmbeispiel präsentieren, welches in der Lage ist, Wertetabellen von Polynomen auszugeben.

Ein häufig gestelltes Problem in Schule und Universität besteht darin, von einem Polynom mehrere Werte auszurechnen, um z.B. die Funktion auf Nullstellen, Wendepunkte usw. zu untersuchen oder diese Funktion grafisch darzustellen. Unser Programm ist in der Lage, beliebige Polynome bis zum 6. Grad zu verarbeiten (in der Praxis hat man es selten mit höheren Graden zu tun). Es können daher Funktionen der Form

y= a6*x↑6 + a5*x↑5 + a4*x↑4 + a3*x↑3 + a2*x↑2 + a1*x + a0

verarbeitet werden. Der Aufruf der Routine geschieht mit

```
SYS 49152,a6,a5,a4,a3,a2,a1,a0,uG,oG,Sw,Ga
```

Dabei bedeuten:
a6–a0: Koeffizienten des Polynoms in absteigender Reihenfolge
uG: Untere Grenze des untersuchten Bereiches
oG: Obere Grenze des untersuchten Bereiches
Sw: Schrittweite zwischen je zwei Werten
Ga: Geräteadresse (3=Bildschirm, 4= Drucker)

Das Programm gibt x- und y-Werte jeweils auf zwei Nachkommastellen gerundet aus. Um die Anwendung zu verdeutlichen, hier ein Beispiel. Es sollen Funktionswerte des Polynoms

y= 3*x↑3 4*x↑2 + 2.5*x –1.6

im Bereich von –3 bis +3 berechnet werden. Die Schrittweite soll 0.5 betragen, die Ausgabe auf dem Bildschirm erfolgen. Der Aufruf der Routine würde also mit

```
SYS 49152,0,0,0,3,-4,2.5,-1.6,-3,3,0.5,3
```

erfolgen. Man erhielte damit folgende Ausgabe:

X	Y
3	126.1
2.5	79.73
2	46.6
1.5	24.48
1	11.1
0.5	5.48
0	1.6
0.5	0.98
1	0.1
1.5	3.28
2	11.4
2.5	26.53
3	50.9

Hier jedoch nun der Quelltext dieses universell einsetzbaren Programms:

Listing: »wertetabelle«

```
10:     c000        -;wertetabelle
11:     c000        -          .ba $c000
13:                 - ;
20:     aefd        -          .eq komma      =    $aefd
21:     ad8a        -          .eq vareal     =    $ad8a
22:     bbd4        -          .eq facmem     =    $bbd4
23:     ab1e        -          .eq stringout=      $ab1e
24:     c400        -          .eq xwert      =    $c400
26:     c405        -          .eq obere      =    xwert + 5
27:     c40a        -          .eq schritt    =    xwert +10
28:     b79e        -          .eq getbyte    =    $b79e
30:     e059        -          .eq polynom    =    $e059
31:     ba28        -          .eq mult       =    $ba28
32:     b849        -          .eq plus05     =    $b849
33:     bccc        -          .eq int        =    $bccc
34:     bafe        -          .eq divid      =    $bafe
35:     ffd2        -          .eq bsout      =    $ffd2
36:     bba2        -          .eq memfac     =    $bba2
38:     e50c        -          .eq cursor     =    $e50c
39:     bddd        -          .eq facstr     =    $bddd
40:     b867        -          .eq plus       =    $b867
41:     bc5b        -          .eq cmpfac     =    $bc5b
42:     00d6        -          .eq zeile      =    $d6
43:     ffba        -          .eq setpar     =    $ffba
44:     ffbd        -          .eq setname    =    $ffbd
45:     ffc0        -          .eq open       =    $ffc0
46:     ffcc        -          .eq clrch      =    $ffcc
47:     ffc3        -          .eq close      =    $ffc3
48:     ffc9        -          .eq chkout     =    $ffc9
49:     00fa        -          .eq laenge     =    $fa
89:                 - ;
90:                 - ; koeffizienten holen
91:                 - ; ==================
92:                 - ;
100: c000 20 fd ae-start    jsr  komma       ;prueft auf komma
101: c003 20 8a ad-         jsr  vareal      ;koeffizient a6 holen
102: c006 a2 b8   -         ldx  #<poly+1    ;abspeichern
103: c008 a0 c1   -         ldy  #>poly+1
103: c00a 20 d4 bb-         jsr  facmem
104: c00d 20 fd ae-         jsr  komma       ;prueft auf komma
105: c010 20 8a ad-         jsr  vareal      ;koeffizient a5 holen
106: c013 a2 bd   -         ldx  #<poly+6    ;abspeichern
107: c015 a0 c1   -         ldy  #>poly+6
```

```
c017 20 d4 bb-          jsr  facmem
c01a 20 fd ae-          jsr  komma        ;prueft auf komma
c01d 20 8a ad-          jsr  vareal       ;koeffizient a4 holen
c020 a2 c2   -          ldx  #<poly+11 ;abspeichern
c022 a0 c1   -          ldy  #>poly+11
c024 20 d4 bb-          jsr  facmem
c027 20 fd ae-          jsr  komma        ;prueft auf komma
c02a 20 8a ad-          jsr  vareal       ;koeffizient a3 holen
c02d a2 c7   -          ldx  #<poly+16 ;abspeichern
c02f a0 c1   -          ldy  #>poly+16
c031 20 d4 bb-          jsr  facmem
c034 20 fd ae-          jsr  komma        ;prueft auf komma
c037 20 8a ad-          jsr  vareal       ;koeffizient a2 holen
c03a a2 cc   -          ldx  #<poly+21 ;abspeichern
c03c a0 c1   -          ldy  #>poly+21
c03e 20 d4 bb-          jsr  facmem
c041 20 fd ae-          jsr  komma        ;prueft auf komma
c044 20 8a ad-          jsr  vareal       ;koeffizient a1 holen
c047 a2 d1   -          ldx  #<poly+26 ;abspeichern
c049 a0 c1   -          ldy  #>poly+26
c04b 20 d4 bb-          jsr  facmem
c04e 20 fd ae-          jsr  komma        ;prueft auf komma
c051 20 8a ad-          jsr  vareal       ;koeffizient a0 holen
c054 a2 d6   -          ldx  #<polx+31 ;abspeichern
c056 a0 c1   -          ldy  #>poly+31
c058 20 d4 bb-          jsr  facmem
             -   ;
             -   ;start- und endwert holen
             -   ;========================
             -   ;
c05b 20 fd ae-          jsr  komma        ;prueft auf komma
c05e 20 8a ad-          jsr  vareal       ;untere grenze holen
c061 a2 00   -          ldx  #<xwert      ;abspeichern
c063 a0 c4   -          ldy  #>xwert      ;(=x-startwert)
c065 20 d4 bb-          jsr  facmem
c068 20 fd ae-          jsr  komma        ;prueft auf komma
c06b 20 8a ad-          jsr  vareal       ;obere grenze holen
c06e a2 05   -          ldx  #<obere      ;abspeichern
c070 a0 c4   -          ldy  #>obere
c072 20 d4 bb-          jsr  facmem
             -   ;
             -   ;schrittweite holen
             -   ;==================
             -   ;
c075 20 fd ae-          jsr  komma        ;prueft auf komma
c078 20 8a ad-          jsr  vareal       ;schrittweite holen
```

```
c07b a2 0a    -           ldx  #<schritt  ;abspeichern
c07d a0 c4    -           ldy  #>schritt
c07f 20 d4 bb-            jsr  facmem
              - ;
              - ;sekundaergeraet holen
              - ;=====================
              - ;
c082 20 fd ae-            jsr  komma      ;prueft auf komma
c085 20 9e b7-            jsr  getbyte    ;geraetenummer holen
c088 e0 03    -           cpx  #03        ;bildschirm
c08a f0 03    -           beq  weiter     ;ja, kein "open"
c08c 20 0c c1-            jsr  offen      ;kanal oeffnen
              - ;
              - ;titelkopf ausgeben
              - ;==================
              - ;
c08f a9 5f    -weiter     lda  #<String1
c091 a0 c1    -           ldy  #>string1  ;1. zeile ausgeben
c093 20 1e ab-            jsr  stringout
c096 a9 87    -           lda  #<string2  ;2. zeile ausgeben
c098 a0 c1    -           ldy  #>string2
c09a 20 1e ab-            jsr  stringout
              - ;
              - ;schleife zur polynomberechnung
              - ;==============================
              - ;
c09d a9 0d    -loop       lda  #13        ;return ausgeben
c09f 20 d2 ff-            jsr  bsout
c0a2 a9 20    -           lda  #32        ;leerzeichen
c0a4 20 d2 ff-            jsr  bsout      ;ausgeben
c0a7 a9 af    -           lda  #<string3
c0a9 a0 c1    -           ldy  #>string3  ;ausgabezeile
c0ab 20 1e ab-            jsr  stringout  ;drucken
c0ae a9 00    -           lda  #<xwert
c0b0 a0 c4    -           ldy  #>xwert    ;fac mit aktuellem
c0b2 20 a2 bb-            jsr  memfac     ;x-wert laden
c0b5 20 45 c1-            jsr  round      ;runden
c0b8 20 dd bd-            jsr  facstr     ;in string umwandeln
c0bb 20 25 c1-            jsr  strout     ;und ausgeben
c0be a9 00    -           lda  #<xwert    ;x-wert in
c0c0 a0 c4    -           ldy  #>xwert    ;fac laden
c0c2 20 a2 bb-            jsr  memfac
c0c5 a9 b7    -           lda  #<poly
c0c7 a0 c1    -           ldy  #>poly
c0c9 20 59 e0-            jsr  polynom    ;polynom berechnen
c0cc 20 45 c1-            jsr  round      ;und runden
```

```
                -  ;
                -  ;ausgabe von y-wert
                -  ;=================
                -  ;
c0cf 20 dd bd-          jsr  facstr     ;in string wandeln
c0d2 20 25 c1-          jsr  strout     ;und ausgeben
c0d5 a9 0d   -          lda  #13
c0d7 20 d2 ff-          jsr  bsout      ;return ausgeben
                -  ;
                -  ;naechsten x-wert berechnen
                -  ;=========================
                -  ;
c0da a9 00   -          lda  #<wert     ;alten x-wert
c0dc a0 c4   -          ldy  #>xwert    ;laden
c0de 20 a2 bb-          jsr  memfac
c0e1 a9 0a   -          lda  #<schritt  ;schrittweite
c0e3 a0 c4   -          ldy  #>schritt  ;addieren
c0e5 20 67 b8-          jsr  plus
c0e8 a9 05   -          lda  #<obere    ;obere grenze
c0ea a0 c4   -          ldy  #>obere    ;erreicht
c0ec 20 5b bc-          jsr  cmpfac
c0ef c9 01   -          cmp  #01        ;ja, ende
c0f1 f0 0a   -          beq  ende
c0f3 a2 00   -          ldx  #<xwert    ;nein, naechsten
c0f5 a0 c4   -          ldy  #>xwert    ;x-wert
c0f7 20 d4 bb-          jsr  facmem     ;berechnen
c0fa 4c 9d c0-          jmp  loop
c0fd a9 87   -ende      lda  #<string2
c0ff a0 c1   -          ldy  #>string2
c101 20 1e ab-          jsr  stringout  ;linie ausgeben
c104 20 cc ff-          jsr  clrch      ;kanal schliessen
c107 a9 01   -          lda  #01        ;file schliessen
c109 4c c3 ff-          jmp  close
                -  ;
                -  ;kanal fuer drucker oeffnen
                -  ;=========================
                -  ;
c10c e0 04   -offen     cpx  #04        ;geraetenummer 4
c10e d0 14   -          bne  off        ;nein, zurueck
c110 a9 01   -          lda  #01        ;filenummer
c112 a0 00   -          ldy  #00        ;sekundaeradresse
c114 20 ba ff-          jsr  setpar     ;parameter setzen
c117 a9 00   -          lda  #00        ;keinen namen
c119 20 bd ff-          jsr  setname    ;setzen
c11c 20 c0 ff-          jsr  open       ;file oeffnen
c11f a2 01   -          ldx  #01        ;ausgabe auf
```

```
261: c121 20 c9 ff-          jsr  chkout     ;drucker legen
262: c124 60       -off      rts
263:               -  ;
264:               -  ;string formatiert ausgeben
265:               -  ;==========================
266:               -  ;
267: c125 20 87 b4-strout    jsr  $b487      ;parameter holen
268: c128 20 a6 b6-          jsr  $b6a6      ;stringverwaltung
269: c12b 85 fa    -         sta  laenge     ;laenge merken
270: c12d 20 24 ab-          jsr  stringout+6  ;string ausgeben
271: c130 a9 10    -         lda  #16        ;anzahl der
272: c132 38       -         sec             ;leerzeichen
273: c133 e5 fa    -         sbc  laenge     ;berechen
274: c135 aa       -         tax
275: c136 a9 20    -         lda  #32        ;code fuer space
276: c138 20 d2 ff-l1        jsr  bsout      ;ausgeben
277: c13b ca       -         dex
278: c13c d0 fa    -         bne  l1
279: c13e a9 af    -         lda  #<stringg3  ;"↑" ausgeben
280: c140 a0 c1    -         ldy  #string3
281: c142 4c 1e ab-          jmp  stringout
282:               -  ;
303:               -  ;rundung auf zwei nachkommastellen
304:               -  ;=================================
305:               -  ;
306:               -  ;fac=(int(fac*100+0.5))/100
307:               -  ;
308: c145 a9 b2    -round    lda  #<hundert
309: c147 a0 c1    -         ldy  #>hundert
310: c149 20 28 ba-          jsr  mult       ;fac=fac*100
311: c14c 20 49 b8-          jsr  plus05     ;fac=fac+0.5
312: c14f 20 cc bc-          jsr  int        ;fac=int(fac)
312: c152 a5 66    -         lda  $66        ;vorzeichen retten
312: c154 58       -         pha
313: c155 20 fe ba-          jsr  divid      ;fac=fac/10
314: c158 20 fe ba-          jmp  divid      ;fac=fac/10
315: c15b 68       -         pla             ;vorzeichen
316: c15c 85 66    -         sta $66         ;wieder holen
317: c15e 60       -         rts
315:               -  ;
500: c15f 0d       -string1  .by 13
550: c160 20 5e 20-.tx" |         x        |        y        |
600: c186 00       -         .by 0
650: c187 0d       -string2  .by 13
700: c188 20 a3 a3-.tx"---------------------------------------
750: c1ae 00       -         .by 0
```

```
800: c1af 5e 20    -string3  .tx "↑"
850: c1b1 00       -         .by 0
900: c1b2 87 48 00-hundert   .by 135,72,0,0,0
950:               - ;
960: c1b7 06       -poly     .by 6
```

Nach dem Aufruf der Routine werden zunächst die Parameter aus dem Basic-Text geholt und die Polynomtabelle angelegt. Falls die Geräteadresse 4 beträgt, wird die Ausgabe auf den Drucker gelenkt. Nun wird der Titelkopf ausgegeben und anschließend in die eigentliche Berechnungsschleife gesprungen. Hier wird zunächst der jeweilige X-Wert gerundet und ausgegeben. Dann kann der y-Wert berechnet, ebenfalls gerundet und ausgegeben werden. Die Ausgabe der Werte erfolgt jeweils in Strings. Der Grund hierfür ist, daß bei der Umwandlung vom Fließkomma- ins Stringformat sehr kleine Zahlen automatisch ins Exponentialformat umgewandelt werden. Für Sie ist es sicherlich angenehmer, statt einer 0.0000001 eine 1E-7 auf dem Bildschirm zu haben. Am Ende der Schleife wird geprüft, ob die obere Grenze erreicht wurde. Ist dies der Fall, wird der Druckerkanal geschlossen und das Programm beendet. Sonst wird der neue X-Wert bestimmt, indem zu dem alten die Schrittweite addiert wird. Mit dieser neuen Zahl wird dann die Polynomberechnung von vorne gestartet.

Diese Routine ist natürlich noch verbesserbar. So könnten Sie z.B. auch eine Ausgabe in ein Diskettenfile zulassen, um die Wertetabelle mit einem Textprogramm weiterverarbeiten zu können, oder Polynome noch höheren Grades zulassen. Der Phantasie sind hier keine Grenzen gesetzt.

3.2 Aufbau der indizierten Variablen (Arrays)

Die indizierten Variablen (Arrays oder auch Felder) unterscheiden sich von den »einfachen« dadurch, daß eine Variable nicht nur einen Wert, sondern eine ganze Tabelle davon enthält. Je mehr Dimensionen vorkommen, desto komplizierter ist der Aufbau der Variablen. Unabhängig vom Variablentyp beinhaltet jedes Feld den sogenannten Header, den man als Kopf des Arrays interpretieren kann. Ein solcher Header ist wie folgt aufgebaut: Die ersten beiden Bytes enthalten die Buchstaben des Namens der Variable, wie dies bei den einfachen Variablen auch der Fall ist. Der Variablentyp wird auch hier durch Setzen eines oder beider 7. Bits der Namenszeichen festgelegt:

Variablentyp	Bit 7 in Byte 1	Byte 2 des Headers
Integer	gesetzt	gesetzt
Fließkomma	nicht gesetzt	nicht gesetzt
String	nicht gesetzt	gesetzt

Der Variablentyp »Funktion« existiert nicht. Anschließend an diese beiden Bytes folgen zwei, welche die Länge des Arrays inklusive des Kopfes angeben. Zunächst kommt das Lowbyte, dann das Highbyte. Natürlich kann man die größtmögliche Länge von 65535 Byte in der Praxis nicht ausnutzen. Das 5. Byte gibt die Anzahl der Dimensionen an. Hier wären also bis zu 255 (!) verwendbar, aber auch hierbei handelt es sich um einen rein theoretischen Wert. Von Byte 6 an folgen für jede Dimension zwei Byte, die die Anzahl der letzten genannten Dimension angeben. Dabei kommt jeweils zuerst das Highbyte, dann das Lowbyte. Man muß hierbei beachten, daß die Zahl der vorhandenen Elemente immer um eins größer ist, als sie vorher dimensioniert wurde, z.B. ergibt das Feld FR(3) folgende Elemente:

FR(0), FR(1), FR(2), FR(3)

Hier nun ein Beispiel für die Belegung der Header-Bytes ab der Position 6 bei Dimensionierung des Feldes FR(3,2,4):

Byte 6:	0
Byte 7:	5 letzte Dimension plus eins
Byte 8:	0
Byte 9:	3 vorletzte Dimension plus eins
Byte 10:	0
Byte 11:	4 vorvorletzte Dimension (=erste) plus eins

Die Länge des Arraykopfes ist also nicht konstant, sondern hängt von der Anzahl der Dimensionen ab. Bei n Dimensionen kann man die Länge mit der Formel

Länge = 2*n + 5

berechnen. Hier nun die Darstellung des Headers für ein zweidimensionales Feld:

1. Byte	2. Byte	3. Byte	4. Byte	5. Byte	6. Byte	7. Byte	8. Byte	9. Byte
Erstes Zeichen (+Bit 7)	Zweites Zeichen (+Bit 7)	Low-byte	High-byte	Zahl der Dimensionen (2)	High-byte	Low-byte	High-byte	Low-byte
Variablenname		Arraylänge inklusive Header			Elementezahl der 2. Dimension		Elementezahl der 1. Dimension	

Anschließend an den Header folgen die einzelnen Arrayelemente. Im Gegensatz zu den einfachen Variablen unterscheiden sie sich hier in ihrer Länge.

3.2.1 Das Arrayelement vom Typ INTEGER

Hatten wir uns bei den nichtindizierten Variablen noch darüber beklagt, daß sie künstlich auf eine Länge von 7 Byte aufgeblasen wurden, werden wir hier positiv überrascht. Ein Element ist genau so lang, wie es sein muß, nämlich zwei Byte:

1. Byte	2. Byte
High-byte	Low-byte
Integerwert	

Im Gegensatz zu den einfachen Variablen lohnt es sich also durchaus, ein Feld vom Typ Integer zu verwenden. Den genauen Platzgewinn gegenüber den Fließkommaelementen kann man wie folgt berechnen:

Man muß das Produkt der Anzahl aller Dimensionen mit der Länge eines Elements bilden. Zusätzlich muß noch die Länge des Headers hinzuaddiert werden. Wenn wir also z.B. ein zweidimensionales Feld der Form FR(10,20) haben, können wir den Platzbedarf eines Fließkomma-Arrays mit der Formel

Länge = 2*2 + 5 + (10+1)*(20+1)*5 = 1164 Byte

Bei den Integerzahlen werden jedoch nur zwei Byte pro Element benötigt:

Länge = 2*2 + 5 + (10+1)*(20+1)*2 = 471 Byte

Der Platzgewinn wirkt sich um so drastischer aus, je mehr Dimensionen und Elemente vorhanden sind.

3.2.2 Das Arrayelement vom Typ FLIESSKOMMA

Ein Fließkommaelement ist genauso aufgebaut, wie wir es auch schon von den nichtindizierten Variablen kennen, nämlich im Format von Exponent und Mantisse. Daher können (natürlich) alle bisher besprochenen Routinen des Interpreters auch bei den Feldern benutzt werden.

1. Byte	2. Byte	3. Byte	4. Byte	5. Byte
Exponent +129 ≡	1. Bit 7 = Vorzeichen	2.	3.	4.
	Mantissenbytes			

3.2.3 Das Arrayelement vom Typ STRING

Auch den Aufbau des Stringfeldes kennen wir schon. Es enthält einen Descriptor, der drei Byte lang ist und einen Zeiger auf die Zeichenkette darstellt. Diese werden normalerweise genau wie bei den einfachen Variablen von der Adresse in $37/$38 herunter angelegt.

1. Byte	2. Byte	3. Byte
String-länge	Low-byte	High-byte
	Stringadresse	

3.2.4 Suchen/Anlegen eines Arrayelementes

Wie auch bei den nichtindizierten Variablen stellt der Interpreter eine Routine hierfür zur Verfügung. Diese ist jedoch darauf ausgerichtet, ihre Informationen aus dem Basic-Text zu holen. Das Problem ist nun, daß die Dimensionen des Arrays nicht in Speicherzellen zwischengespeichert werden können, da ihre Anzahl und damit auch die Zahl der benötigten Speicherplätze variabel sind. Die Interpreter-Routine behilft sich nun damit, diese Dimensionen auf den Stapel zu schieben und bei der Verarbeitung wieder herunterzuholen. Hier sieht man nun auch, daß man keineswegs 255 Dimensionen verwenden kann, da der Stapel natürlich keine 512 Byte (die Dimensionen werden in Low- und Highbyte aufgeteilt) aufnehmen kann. Entsprechend kompliziert ist die Einsprungroutine in Maschinensprache. Es gibt sogar Autoren, die der Meinung sind, man sollte lieber eigene Routinen zur Suche eines Elementes verwenden. Dieser Aufwand steht aber in keinem Verhältnis zu den Vorbereitungen des Einsprungs in die Interpreterroutine. Zunächst müssen, wie bei den einfachen Variablen, die Flags für den Variablentyp sowie der Variablenname gesetzt werden. Dann werden die Dimensionen hintereinander mit Hilfe des Stackpointers gestapelt. Die Zahl der Dimensionen wird ebenfalls übergeben. Dann kann man mit

```
JSR $B20E
```

in die Routine einspringen. Die Adresse des Arrayelementes erhält man in den Speicherstellen $47 (Lowbyte) und $48 (Highbyte) zurück. Da der Aufwand mit den Dimensionen steigt, möchte ich hier nur die Einsprünge für ein- sowie für zweidimensionale Felder aufführen. Zunächst wollen wir das Element 300 des Feldes FR suchen (FR(300)):

```
     JSR HOLE     ;Routine muss unbedingt mit
     LDA $47      ;JSR angesprungen werden (Stack!)
     LDY $48      ;Ergebnis in Akku und Y-Register
     RTS

HOLE LDA #"F"     ;ersten Buchstaben des Namens
     STA $45      ;setzen
```

```
     LDA #"R"     ;zweiten Buchstaben des Namens
     STA $46      ;setzen
     LDA #00
     STA $0E      ;Integer-Flag
     STA $0D      ;String-Flag
     STA $0C      ;DIM-Flag löschen
    (JSR INT)     ;nur bei Integer-Arrays ausfuehren!
    (JSR STR)     ;nur bei String-Arrays ausfuehren!
     LDA $0C
     ORA $0E      ;alle Flags auf Stack
     PHA
     LDA $0D
     PHA
     TSX          ;Stackpointer holen
     LDA $0102,X ;Flags wieder von
     PHA          ;Stapel holen
     LDA $0101,X ;und sofort wieder
     PHA          ;stapeln
     LDA #01      ;High-Byte von 300
     STA $0102,X ;in Stapel-Luecke
     LDA #44      ;Low-Byte von 300
     STA $0101,X ;in Stapel-Luecke
     LDY #01      ;Zahl der Dimensionen=1
     STY $0B      ;uebergeben
     JMP $B20E    ;Einsprung in Interpreterroutine
     ;
INT  LDA #$80     ;7.Bit von erstem
     ORA $45      ;Buchstaben setzen
     STA $45
     LDA #$80     ;Integerflag
     STA $0E
L1   ORA $46      ;und 7.Bit von zweitem
     STA $46      ;Buchstaben setzen
     RTS
     ;
STR  LDA #$FF     ;Stringflag setzen
     STA $0D
     LDA #$80     ;7.Bit von zweitem
     BMI L1       ;Buchstaben setzen
```

Sicherlich werden Sie sich fragen, was die undurchsichtigen Stack-Befehle bewirken. Nun, zunächst werden die Flags mit den Push-Befehlen normal gestapelt. Dann werden sie heruntergeholt, ohne den Stackpointer zu verändern, und wiederum gestapelt. Zu diesem Zeitpunkt sind also die beiden Bytes je doppelt vorhanden. Die zuerst gestapelten Bytes werden jedoch anschließend durch die beiden Bytes der Dimension ersetzt, so daß sich nun die Dimension vor den Flags auf dem Stapel befindet. Der Sinn dieser Methode besteht darin, daß man also

in jedem Fall zuerst die Flags vom Stapel holen kann. Hätte man die Dimension »normal« gestapelt, wären die Flags unerreichbar auf dem Stack nach oben geschoben worden. Bei mehreren Dimensionen hat man also am Ende der Stapelung folgende Reihenfolge auf dem Stack:

1.) 1. Dimension
2.) 2. Dimension

n.) n. Dimension
n+1). Flags

Mit den PLA-Befehlen kann man nun zuerst die Flags und dann die Dimensionen in absteigender Reihenfolge vom Stack holen. Genau diese Reihenfolge aber benötigt die Interpreterroutine. Hier jetzt unser zweidimensionales Beispiel, wo wir das jeweils zehnte Element jeder Dimension des Feldes »G$« suchen wollen (G$(10,10)):

```
     JSR HOLE      ;Routine muss unbedingt mit
     LDA $47       ;JSR angesprungen werden (Stack!)
     LDY $48       ;Ergebnis in Akku und Y-Register
     RTS

HOLE LDA #"G"      ;ersten Buchstaben des Namens
     STA $45       ;setzen
     LDA #0        ;kein zweiter Buchstabe
     STA $46       ;vorhanden
     LDA #00
     STA $0E       ;Integer-Flag
     STA $0D       ;String-Flag
     STA $0C       ;DIM-Flag loeschen
    (JSR INT)      ;nur bei Integer-Arrays ausfuehren!
     JSR STR       ;Variablentyp String setzen
     LDA $0C
     ORA $0E       ;alle Flags auf Stack
     PHA
     LDA $0D
     PHA
L2   TSX           ;Stackpointer holen
     LDA $0102,X   ;Flags wieder von
     PHA           ;Stapel holen
     LDA $0101,X   ;und sofort wieder
     PHA           ;stapeln
     LDA #00       ;Highbyte von 10 (erste Dimension)
     STA $0102,X   ;in Stapel-Luecke
     LDA #10       ;Lowbyte von 10 (erste Dimension)
L3   STA $0101,X   ;in Stapel-Luecke
     TSX           ;Stackpointer holen
     LDA $0102,X   ;Flags wieder von
     PHA           ;Stapel holen
```

```
        LDA $0101,X ;und sofort wieder
        PHA         ;stapeln
        LDA #00     ;Highbyte von 10 (zweite Dimension)
        STA $0102,X ;in Stapel-Luecke
        LDA #10     ;Lowbyte von 10 (zweite Dimension)
        STA $0101,X ;in Stapel-Luecke
        LDY #02     ;Anzahl der Dimensionen
        STY $0B     ;uebergeben
        JMP $B20E   ;Einsprung in Interpreterroutine
        ;
INT     LDA #$80    ;7.Bit von erstem
        ORA $45     ;Buchstaben setzen
        STA $45
        LDA #$80    ;Integerflag
        STA $0E
L1      ORA $46     ;und 7.Bit von zweitem
        STA $46     ;Buchstaben setzen
        RTS
        ;
STR     LDA #$FF    ;Stringflag setzen
        STA $0D
        LDA #$80    ;7.Bit von zweitem
        BMI L1      ;Buchstaben setzen
```

Hierbei erkennt man, wie bei weiteren Dimensionen verfahren werden muß: Der Teil von L2 bis L3 muß für jede weitere Dimension wiederholt werden, wobei jeweils die Zahl der letzten Dimension des gesuchten Elementes gestapelt wird. Natürlich muß auch die Zahl der Dimensionen (Y-Register) erhöht werden. Das Problem ist, daß man den Bereich von L2 bis L3 nicht unterprogrammäßig anspringen kann, da sonst der Stackpointer verändert würde und unkorrekte Stapelungen die Folge wären. Man muß diesen Teil also jedesmal wieder anhängen.

Falls das gesuchte Array nicht gefunden wird, wird genau wie bei den einfachen Variablen ein neues an die bereits bestehenden angehängt. Dabei muß jedoch beachtet werden, daß genau wie beim Aufruf eines undimensionierten Feldes von Basic aus nur ein eindimensionales Feld mit 11 Elementen angelegt werden kann. Im anderen Fall wird mit der Fehlermeldung »?BAD SUBSCRIPT ERROR« abgebrochen. Möchte man also größere Felder einrichten, muß das DIM-Flag gesetzt werden. Dafür muß man den Speicher $0C mit einer 65 statt einer 0 am Anfang der Routine versorgen:

```
..........      ;Anfang uebernehmen
;
HOLE    ........;uebernehmen
        LDA #65 ;DIM-Flag setzen
        STA $0C
        LDA #00 ;Integer- und
        STA $0D ;String-Flag
```

```
      STA $0E ;loeschen
..............;Rest uebernehmen
```

Hiermit wird der Interpreterroutine vorgegaukelt, der Einsprung käme von dem Basic-Befehl DIM, der ja bekanntlich dazu dient, Arrays zu dimensionieren. Falls Sie einmal die gesamte Routine studieren möchten, müßten Sie das ROM-Listing von $B07E bis $B37C durchlesen.

3.2.5 Bubblesort in Maschinensprache

Eines der größten Probleme z.B. einer Dateiverwaltung besteht darin, ein umfangreiches Feld in annehmbarer Zeit zu sortieren. Jeder Basic-Programmierer kann davon ein Lied singen. Wir wollen hier deshalb einen einfachen Sortieralgorithmus in Maschinensprache übertragen und sehen, welchen Geschwindigkeitsgewinn wir erhalten. Unter einem »Bubblesort« versteht man in Basic die folgenden Zeilen, wobei die Zahl der Elemente »n« ist und der Feldname FR sein soll:

```
1 FOR I = 1 TO N : FLAG = 0
2 FOR J = N TO I STEP -1
3 IF FR(J-1)  FR(J) THEN K=FR(J):FR(J)=FR(J-1):FR(J-1)=K:FLAG=1
4 NEXT J
5 IF FLAG = 0 THEN 7
6 NEXT I
7 END
```

Durch Ersatz des Feldes FR durch FR$ bzw. FR% kann man auch Strings und Integer-Variablen sortieren. In Maschinensprache muß man jedoch für jeden Variablentyp eine eigene Version verwenden.

Das Problem dieses Algorithmus besteht darin, daß zwei ineinander verschachtelte Schleifen durchlaufen werden müssen, was zu einem etwa quadratischen Anstieg der Sortierzeit führt. Bei einer Verdoppelung der Elemente muß man daher mit vierfachem Zeitverbrauch rechnen, bei einer Verzehnfachung wächst die Zeit um den Faktor 100 ! In Basic wird man daher kaum mit Bubblesort arbeiten, wenn größere Mengen von Daten zu sortieren sind. Wie wir aber sehen werden, sind die Ergebnisse in Maschinensprache durchaus akzeptabel.

3.2.5.1 Bubblesort für Integer-Variablen

Wir starten mit der einfachsten Version: Hier müssen nur jeweils Low- und Highbyte verglichen werden. Das einfache Basic-Programm sieht in Maschinensprache so aus:

Listing: »bubbleinteger«

```
0:   c200        -  ;bubblesort-integer
1:   c200        -          .ba $c200
2:               -  ;
10:  aefd        -          .eq chkcom   =    $aefd
```

```
b08b          -         .eq variab    =     $b08b
00ab          -         .eq flag      =     $ab
005f          -         .eq array     =     $5f
00a7          -         .eq i         =     $a7
00a9          -         .eq j         =     $a9
008b          -         .eq j1        =     $8b
008d          -         .eq n         =     $8d
              -  ;
c200 20 fd ae-start     jsr  chkcom
c203 20 8b b0-          jsr  variab           ;startadresse
c206 a5 5f    -         lda  array            ;des feldes holen
c208 a0 02    -         ldy  #02
c20a 18       -         clc
c20b 71 5f    -         adc  (array),y        ;+arraylaenge
c20d 85 8d    -         sta  n                ;=ende des arrays
c20f c8       -         iny
c210 a5 60    -         lda  array+1
c212 71 5f    -         adc  (array),y
c214 85 8e    -         sta  n+1
c216 a5 8d    -         lda  n                ;zeiger auf
c218 38       -         sec                   ;letztes
c219 e9 02    -         sbc  #02              ;arrayelement
c21b 85 8d    -         sta  n                ;setzen
c21d b0 02    -         bcs  loop1            ;(=ende-5 bytes)
c21f c6 8e    -         dec  n+1
              -  ;
c221 a5 5f    -loop1    lda  array
c223 18       -         clc                   ;start auf
c224 69 07    -         adc  #07              ;erstes
c226 85 a7    -         sta  i                ;arrayelement
c228 a5 60    -         lda  array+1          ;setzen
c22a 69 00    -         adc  #00              ;(headerlaenge
c22c 85 a8    -         sta  i+1              ;=7 bytes)
              -  ;
c22e a0 00    -loopi    ldy  #00              ;schleife i
c230 84 ab    -         sty  flag
c232 a5 8d    -         lda  n                ;fl = 0
c234 85 a9    -         sta  j
c236 a5 8e    -         lda  n+1              ;j=n
c238 85 aa    -         sta  j+1
              -  ;
c23a a5 a9    -loopj    lda  j                ;schleife j
c23c 38       -         sec
c23d e9 02    -         sbc  #02              ;j1=j-1
c23f 85 8b    -         sta  j1
c241 a5 aa    -         lda  j+1              ;feld%(j-1)
```

```
c243 e9 00  -           sbc  #00
c245 85 8c  -           sta  j1+1
c247 a0 00  -           ldy  #00           ;hi-byte feld%(j)
c249 b1 a9  -           lda  (j),y         ;=feld%(j-1)
c24b d1 8b  -           cmp  (j1),y        ;ja, dann weiter
c24d 90 07  -           bcc  tausch
c24f c8     -           iny                ;lowbyte feld%(j)
c250 b1 a9  -           lda  (j),y         ;=feld%(j-1)
c252 d1 8b  -           cmp  (j1),y        ;ja, dann weiter
c254 b0 0f  -           bcs  weiter
            - ;
c256 84 ab  -tausch     sty  flag          ;flag setzen
c258 b1 a9  -l1         lda  (j),y
c25a aa     -           tax                ;feld(j) mit
c25b b1 8b  -           lda  (j1),y
c25d 91 a9  -           sta  (j),y         ;feld(j-1)
c25f 8a     -           txa
c260 91 8b  -           sta  (j1),y        ;tauschen
c262 88     -           dey
c263 10 f3  -           bpl  l1
            - ;
c265 a5 a9  -weiter     lda  j
c267 38     -           sec
c268 e9 02  -           sbc  #02           ;j=j-1
c26a 85 a9  -           sta  j
c26c b0 02  -           bcs  endj
c26e c6 aa  -           dec  j+1
            - ;
c270 c5 a7  -endj       cmp  i             ;j bis auf i
c272 d0 c6  -           bne  loopj         ;heruntergezaehlt
c274 a5 aa  -           lda  j+1           ;wenn nein, mit
c276 c5 a8  -           cmp  i+1           ;schleife j
c278 d0 c0  -           bne  loopj         ;weitermachen
            - ;
c27a a5 ab  -           lda  flag          ;kein tausch,dann
c27c f0 15  -           beq  ende          ;ende
            - ;
c27e a5 a7  -           lda  i
c280 18     -           clc                ;i=i+1
c281 69 02  -           adc  #02
c283 85 a7  -           sta  i
c285 90 02  -           bcc  endi
c287 e6 a8  -           inc  i+1
            - ;
c289 c5 8d  -endi       cmp  n             ;i bis auf n
c28b d0 a1  -           bne  loopi         ;heraufgezaehlt
```

```
105: c28d a5 a8     -          lda  i+1          ;wenn nein, mit
106: c28f c5 8e     -          cmp  n+1          ;schleife i
107: c291 d0 9b     -          bne  loopi        ;weitermachen
108:                - ;
109: c293 60          ende     rts
```

Der Aufruf ist in diesem Fall mit

```
SYS 49664, FR%(0)
```

von Basic aus realisiert worden, wobei FR% der Name des zu sortierenden Arrays ist. Oben haben wir aber auch gesehen, wie das Anfangselement von Maschinensprache aus gesucht und gefunden werden kann.

Man erkennt, daß der eigentliche Vergleich von je zwei Feldelementen in den Zeilen 61 bis 67 durchgeführt wird. Das Programm ist genau analog zum Basic-Programm aufgebaut. Für die Variablen I, J, J-1 und N sind je zwei Byte reserviert worden, so daß beliebig große Arrays sortiert werden können. Falls Sie nur mit Arrays bis zu 256 Elementen arbeiten möchten, können Sie natürlich alle Zwei-Byte-Werte in Ein-Byte-Werte abändern und somit die Geschwindigkeit noch erhöhen. Stellvertretend für alle drei Versionen möchte ich hier kurz den Programmaufbau erläutern, der streng am Basic-Programm orientiert ist. Nachdem die Startadresse des Arrays in \$5F/\$60 zur Verfügung steht, wird mit Hilfe der Arraylänge ein Zeiger hinter das letzte Element berechnet (n). Da wir jedoch immer die Position am Anfang des Elementes benötigen, muß hiervon noch die Feldlänge (bei Integerzahlen = zwei) subtrahiert werden (Zeilen 31–36). Nun wird die I-Schleife eingerichtet. Dazu wird der Zeiger I vor das erste Feldelement gesetzt, was man dadurch erreicht, daß man zur Startadresse des Headers 7 Byte addiert. Wie wir ja oben gesehen haben, ist dies die Headerlänge bei eindimensionalen Feldern (Zeilen 38–44). In den Zeilen 46–51 wird die J-Schleife eingerichtet, indem J gleich N gesetzt wird. Nun wird ein Zeiger auf das Element J-1 gesetzt, indem von dem J-Zeiger die Länge eines Elementes (2 Byte) abgezogen wird. Anschließend werden beide Arrayelemente byteweise verglichen (Zeilen 53–67). In den Zeilen 69–78 wird dann ein eventuell erforderlicher Tausch durchgeführt. Dann wird der J-Zeiger um eins erniedrigt (Zeilen 80–85) und geprüft, ob »J« bis auf »I« heruntergezählt wurde. Ist dies der Fall, wird das Flag geprüft. Falls es gesetzt wurde, ist der Sortiervorgang beendet (Zeilen 87–91). Sonst wird der I-Zeiger erhöht (Zeilen 96–101) und daraufhin überprüft, ob der Endwert »n« erreicht wurde (Zeilen 103–107). Falls dies gegeben ist, kann das Programm beendet werden. Sonst wird in der äußeren Schleife mit neuem I-Wert weitersortiert. Hier noch ein Geschwindigkeitsvergleich zwischen Basic- und Maschinenspracheversion, wobei »n« die Anzahl der Elemente ist:

N	Basic	Maschinensprache
10	1s	0.0s
50	28s	0.1s
100	2min27s	0.7s
200	11min58s	2.7s
500	1h21min21s	13.1s
1000	5h25min44s	53.4s

Die Maschinenroutine ist gut 200- bis 380mal schneller. Bei der Basic-Routine muß man jedoch berücksichtigen, daß ja keine Integer-Routinen existieren und daher alle Werte in Fließkommazahlen umgewandelt werden müssen, bevor sie verglichen werden können.

3.2.5.2 Bubblesort für Fließkommavariablen

Die hierfür benötigte Routine unterscheidet sich von der vorhergehenden dadurch, daß der Vergleich zweier Elemente nun nicht mehr unmittelbar durchgeführt werden kann, sondern im FAC ausgeführt werden muß. Ebenfalls unterschiedlich ist der Abstand zwischen je zwei Elementen und deren Länge. Der Aufbau entspricht jedoch genau dem der vorigen Routine:

Listing: »bubblefliess«

```
0:   c000          -;bubblesort-fließkomma
1:   c000          -          .ba $c000
2:                 -  ;
10:  aefd          -          .eq chkcom   =    $aefd
11:  b08b          -          .eq variab   =    $b08b
12:  bba2          -          .eq memfac   =    $bba2
13:  bc5b          -          .eq cmpfac   =    $bc5b
14:  00ab          -          .eq flag     =    $ab
15:  005f          -          .eq array    =    $5f
16:  00a7          -          .eq i        =    $a7
17:  00a9          -          .eq j        =    $a9
18:  008b          -          .eq j1       =    $8b
19:  008d          -          .eq n        =    $8d
20:                -  ;
21:  c000 20 fd ae-start      jsr  chkcom
22:  c003 20 8b b0-           jsr  variab          ;startadresse
23:  c006 a5 5f    -          lda  array           ;des feldes holen
24:  c008 a0 02    -          ldy  #02
24:  c00a 18       -          clc
25:  c00b 71 5f    -          adc  (array),y       ;+arraylaenge
26:  c00d 85 8d    -          sta  n               ;=ende des arrays
27:  c00f c8       -          iny
28:  c010 a5 60    -          lda  array+1
```

```
c012 71 5f    -          adc   (array),y
c014 85 8e    -          sta   n+1
c016 a5 8d    -          lda   n             ;zeiger auf
c018 38       -          sec                 ;letztes
c019 e9 05    -          sbc   #05           ;arrayelement
c01b 85 8d    -          sta   n             ;setzen
c01d b0 02    -          bcs   loop1         ;(=ende-5 bytes)
c01f c6 8e    -          dec   n+1
              -  ;
c021 a5 5f    -loop1     lda   array
c023 18       -          clc                 ;start auf
c024 69 07    -          adc   #07           ;erstes
c026 85 a7    -          sta   i             ;arrayelement
c028 a5 60    -          lda   array+1       ;setzen
c02a 69 00    -          adc   #00           ;(headerlaenge
c02c 85 a8    -          sta   i+1           ;=7 bytes)
              -  ;
c02e a0 00    -loopi     ldy   #00           ;schleife i
c030 84 ab    -          sty   flag
c032 a5 8d    -          lda   n             ;fl = 0
c034 85 a9    -          sta   j
c036 a5 8e    -          lda   n+1           ;j=n
c038 85 aa    -          sta   j+1
              -  ;
c03a a5 a9    -loopj     lda   j             ;schleife j
c03c 38       -          sec
c03d e9 05    -          sbc   #05           ;j1=j-1
c03f 85 8b    -          sta   j1
c041 aa       -          tax
c042 a5 aa    -          lda   j+1           ;feld(j-1)
c044 e9 00    -          sbc   #00
c046 85 8c    -          sta   j1+1          ;nach fac
c048 a8       -          tay
c049 8a       -          txa                 ;holen
:  c04a 20 a2 bb-       jsr  memfac
c04d a5 a9    -          lda   j             ;und mit feld(j)
c04f a4 aa    -          ldy   j+1
c051 20 5b bc-           jsr   cmpfac        ;vergleichen
c054 30 11    -          bmi   weiter        ;feld(j-1)(j)
              -  ;
c056 e6 ab    -tausch    inc   flag          ;flag setzen
c058 a0 04    -          ldy   #04
c05a b1 a9    -l1        lda   (j),y
c05c aa       -          tax                 ;feld(j) mit
c05d b1 8b    -          lda   (j1),y
c05f 91 a9    -          sta   (j),y         ;feld(j-1)
c061 8a       -          txa
```

```
c062 91 8b  -         sta  (j1),y       ;tauschen
c064 88     -         dey
c065 10 f3  -         bpl  l1
            -  ;
c067 a5 a9  -weiter   lda  j
c069 38     -         sec
c06a e9 05  -         sbc  #05          ;j=j-1
c06c 85 a9  -         sta  j
c06e b0 02  -         bcs  endj
c070 c6 aa  -         dec  j+1
            -  ;
c072 c5 a7  -endj     cmp  i            ;j bis auf i
c074 d0 c4  -         bne  loopj        ;heruntergezaehlt
c076 a5 aa  -         lda  j+1          ;wenn nein, mit
c078 c5 a8  -         cmp  i+1          ;schleife j
c07a d0 be  -         bne  loopj        ;weitermachen
            -  ;
c07c a5 ab  -         lda  flag         ;kein tausch,dann
c07e f0 15  -         beq  ende         ;ende
            -  ;
c080 a5 a7  -         lda  i
c082 18     -         clc               ;i=i+1
c083 69 05  -         adc  #05
c085 85 a7  -         sta  i
c087 90 02  -         bcc  endi
c089 e6 a8  -         inc  i+1
            -  ;
c08b c5 8d  -endi     cmp  n            ;i bis auf n
c08d d0 9f  -         bne  loopi        ;heraufgezaehlt
c08f a5 a8  -         lda  i+1          ;wenn nein, mit
c091 c5 8e  -         cmp  n+1          ;schleife i
c093 d0 99  -         bne  loopi        ;weitermachen
            -  ;
c095 60     -ende     rts
```

Der Aufruf geschieht hier mit

```
SYS 49152,FR(0)
```

wenn FR der Name des Fließkomma-Arrays ist.

Auch hier soll noch ein Zeitvergleich zwischen Basic- und Maschinenroutine angeführt werden:

N	Basic	Maschinenroutine
10	1s	0.0s
50	27s	0.5s
100	2min15s	1.6s
200	10min59s	6.8s
500	1h14min12s	39.3s
1000	4h58min34s	2min35.2s

Man sieht, daß hier der Zeitgewinn wesentlich geringer ist als bei den Integervariablen. Der Grund ist, daß man den umständlichen Vergleich zweier Fließkommazahlen im FAC durchführen muß.

3.2.5.3 Bubblesort für Strings

Der umständlichste Vergleich ist der zweier beliebiger Strings. Der Grund hierfür ist, daß diese kein einheitliches Format haben, sondern eine unterschiedliche Länge aufweisen können. Wir wollen uns einmal an Hand eines Beispiels verdeutlichen, welche Fälle auftreten können und wie sie zu interpretieren sind. Dazu sollen zunächst die Strings

»Commodore« und »Comnodore«

verglichen werden. Wenn man zwei Strings vergleicht, dann muß dies zeichenweise vom Anfang bis Ende geschehen. Die beiden hier betrachteten Strings sind gleichlang und unterscheiden sich im vierten Zeichen. Man muß also offenbar so lange Zeichen für Zeichen vergleichen, bis ein Unterschied feststellbar ist. Hier ist der ASCII-Code von »n« größer als der von »m«, so daß der String »Comnodore« als größer eingestuft werden kann. Falls zwei gleichlange Strings bis zum letzten Zeichen keinerlei Unterschied aufweisen, sind sie offensichtlich gleich.
Problematisch wird die Sache erst bei unterschiedlich langen Strings, wie bei

»Commodore« und »Comno«

Hier ist ein Vergleich nur bis zum einschließlich 5. Zeichen sinnvoll, da man ja nicht weiß, was hinter dem »o« des zweiten Strings folgt. Man kann daher nur die Zeichen 1 bis 5 vergleichen. Da im vierten Zeichen ein Unterschied besteht, ist es auch hier einfach, den größeren String zu finden, nämlich »Comno«. Hier wird deutlich, daß es sehr wichtig ist, zwischen dem längeren und dem größeren String zu unterscheiden.

Der dritte Fall schließlich behandelt zwei Strings, die unterschiedlich lang sind, jedoch bis zur kleinsten gemeinsamen Länge keinen Unterschied aufweisen, wie z.B.

»Commodore« und »Commo«

Hier wird der längere als größer definiert.

Unser Sortierprogramm muß also diese drei Fälle unterscheiden. Zusammenfassend kann man sagen, daß zunächst bis zur Länge des kürzeren Strings verglichen werden muß. Tritt bis dorthin ein Unterschied auf, ist der Fall klar. Ist dies nicht der Fall, muß man die Länge beider Strings vergleichen. Wenn beide Strings gleichlang sind, sind sie auch gleich, während bei unterschiedlicher Länge der längere String als größer definiert wird. Genau diese Forderungen erfüllt das Unterprogramm »VERGLEICH«, das an die Stelle der Vergleichsroutinen des FAC tritt:

Listing: »bubblestrings«

```
c100           -;bubblesort-strings
c100           -          .ba $c100
               -  ;
aefd           -          .eq chkcom   =   $aefd
b08b           -          .eq variab   =   $b08b
00ab           -          .eq flag     =   $ab
005f           -          .eq array    =   $5f
00a7           -          .eq i        =   $a7
00a9           -          .eq j        =   $a9
008b           -          .eq j1       =   $8b
008d           -          .eq n        =   $8d
00fa           -          .eq pj       =   $fa
00fc           -          .eq pj1      =   $fc
008f           -          .eq l        =   $8f
00fe           -          .eq l1       =   $fe
               -  ;
c100 20 fd ae-start       jsr  chkcom
c103 20 8b b0-            jsr  variab           ;startadresse
c106 a5 5f     -          lda  array            ;des feldes holen
c108 a0 02     -          ldy  #02
c10a 18        -          clc
c10b 71 5f     -          adc  (array),y        ;+arraylaenge
c10d 85 8d     -          sta  n                ;=ende des arrays
c10f c8        -          iny
c110 a5 60     -          lda  array+1
c112 71 5f     -          adc  (array),y
c114 85 8e     -          sta  n+1
c116 a5 8d     -          lda  n                ;zeiger auf
c118 38        -          sec                   ;letztes
c119 e9 03     -          sbc  #03              ;arrayelement
c11b 85 8d     -          sta  n                ;setzen
c11d b0 02     -          bcs  loop1            ;(=ende-3 bytes)
c11f c6 8e     -          dec  n+1
               -  ;
c121 a5 5f     -loop1     lda  array
c123 18        -          clc                   ;start auf
```

```
40:  c124 69 07    -         adc  #07         ;erstes
41:  c126 85 a7    -         sta  i           ;arrayelement
42:  c128 a5 60    -         lda  array+1     ;setzen
43:  c12a 69 00    -         adc  #00         ;(headerlaenge
44:  c12c 85 a8    -         sta  i+1         ;=7 bytes)
45:                -  ;
46:  c12e a0 00    -loopi    ldy  #00         ;schleife i
47:  c130 84 ab    -         sty  flag
48:  c132 a5 8d    -         lda  n           ;fl = 0
49:  c134 85 a9    -         sta  j
50:  c136 a5 8e    -         lda  n+1         ;j=n
51:  c138 85 aa    -         sta  j+1
52:                -  ;
53:  c13a a5 a9    -loopj    lda  j           ;schleife j
54:  c13c 38       -         sec
55:  c13d e9 03    -         sbc  #03         ;j1=j-1
56:  c13f 85 8b    -         sta  j1
57:  c141 aa       -         tax
58:  c142 a5 aa    -         lda  j+1         ;feld$(j-1)
59:  c144 e9 00    -         sbc  #00
60:  c146 85 8c    -         sta  j1+1        ;feld$(j) mit
63:  c148 20 7a c1-          jsr  vergleich   ;feld$(j-1) ver-
64:                -  ;                       ;gleichen und ggf
65:                -  ;                       ;tauschen
80:  c14b a5 a9    -weiter   lda  j
81:  c14d 38       -         sec
82:  c14e e9 03    -         sbc  #03         ;j=j-1
83:  c150 85 a9    -         sta  j
84:  c152 b0 02    -         bcs  endj
85:  c154 c6 aa    -         dec  j+1
86:                -  ;
87:  c156 c5 a7    -endj     cmp  i           ;j bis auf i
88:  c158 d0 e0    -         bne  loopj       ;heruntergezaehlt
89:  c15a a5 aa    -         lda  j+1         ;wenn nein, mit
90:  c15c c5 a8    -         cmp  i+1         ;schleife j
91:  c15e d0 da    -         bne  loopj       ;weitermachen
92:                -  ;
93:  c160 a5 ab    -         lda  flag        ;kein tausch,dann
94:  c162 f0 15    -         beq  ende        ;ende
95:                -  ;
96:  c164 a5 a7    -         lda  i
97:  c166 18       -         clc              ;i=i+1
98:  c167 69 03    -         adc  #03
99:  c169 85 a7    -         sta  i
100: c16b 90 02    -         bcc  endi
101: c16d e6 a8    -         inc  i+1
102:               -  ;
```

```
c16f c5 8d   -endi       cmp  n          ;i bis auf n
c171 d0 bb   -           bne  loopi      ;heraufgezaehlt
c173 a5 a8   -           lda  i+1        ;wenn nein, mit
c175 c5 8e   -           cmp  n+1        ;schleife i
c177 d0 b5   -           bne  loopi      ;weitermachen
             -  ;
c179 60      -ende       rts
             -  ;
c17a a0 00   -vergleichldy  #00
c17c b1 a9   -           lda  (j),y
c17e 85 8f   -           sta  l          ;laenge feld$(j)
c180 aa      -           tax
c181 b1 8b   -           lda  (j1),y
c183 85 fe   -           sta  l1         ;laenge$(j-1)
c185 c5 8f   -           cmp  l
c187 90 01   -           bcc  weiter1    ;keinere laenge
c189 8a      -           txa             ;zum vergleich
c18a 8d be c1-weiter1    sta  klein+1
c18d c8      -           iny
c18e b1 a9   -           lda  (j),y
c190 85 fa   -           sta  pj
c192 b1 8b   -           lda  (j1),y     ;stringpointer
c194 85 fc   -           sta  pj1
c196 c8      -           iny             ;setzen
c197 b1 a9   -           lda  (j),y
c199 85 fb   -           sta  pj+1
c19b b1 8b   -           lda  (j1),y
c19d 85 fd   -           sta  pj1+1
             -  ;
c19f a0 00   -compare    ldy  #00
c1a1 b1 fc   -l2         lda  (pj1),y    ;zeichenweise
c1a3 d1 fa   -           cmp  (pj),y     ;vergleichen
c1a5 f0 15   -           beq  weiter2    ;f.$(j)=f.$(j-1)
c1a7 90 1e   -           bcc  weiter3    ;f.$(j).$(j-1)
             -  ;
c1a9 a0 02   -tausch     ldy  #02
c1ab 84 ab   -           sty  flag       ;flag setzen
c1ad b1 8b   -l3         lda  (j1),y
c1af aa      -           tax
c1b0 b1 a9   -           lda  (j),y      ;stringpointer
c1b2 91 8b   -           sta  (j1),y
c1b4 8a      -           txa             ;vertauschen
c1b5 91 a9   -           sta  (j),y
c1b7 88      -           dey
c1b8 10 f3   -           bpl  l3
c1ba 30 0b   -           bmi  weiter3    ;springt immer
             -  ;
```

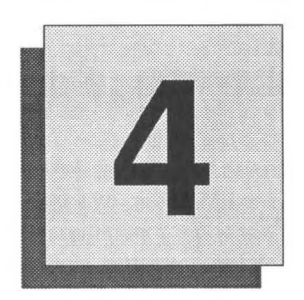
```
150: c1bc c8       -weiter2  iny              ;naechstes zeichen
151: c1bd c0 00    -klein    cpy  #00         ;vergleichen
152: c1bf d0 e0    -         bne  12
153:               -  ;
154: c1c1 a5 8f    -gleich   lda  1           ;der string mit
155: c1c3 c5 fe    -         cmp  11          ;der groesseren
156: c1c5 90 e2    -         bcc  tausch      ;laenge ist  !
157:               -  ;
158: c1c7 60       -weiter3  rts
```

Diese Version wird mit

```
SYS 49408,FR$(0)
```

aufgerufen, wobei FR$ der Name des Stringarrays ist, das sortiert werden soll. Über die Sortierzeiten können hier keine Angaben gemacht werden, da sie sehr stark von der Länge der einzelnen Strings abhängen.

Wer mit den Zeiten des Bubblesort-Algorithmus nicht zufrieden ist oder große Datenmengen sortieren muß, sollte auf den Quicksort-Algorithmus wechseln. Eine Maschinenspracheversion dieses extrem aufwendigen, aber auch schnellen Verfahrens findet sich z.B. im 64'er-Magazin, Ausgabe Dezember 1985. Mit dieser Routine kann man 1000 Strings in unter acht Sekunden sortieren.



Programmierung der HiRes-Grafik

Sicherlich werden Sie sich fragen, warum sich auch dieses Buch mit der Grafikprogrammierung befaßt, obwohl doch seit Einführung des C64 über kein anderes Thema so viel gesprochen und geschrieben wurde, wie gerade über die Grafik.

Der Grund ist sehr einfach: Zum einen ist es immer wieder faszinierend, was ein Rechner der untersten Preisklasse, zu der ja der C64 ohne Zweifel gehört, auf diesem Gebiet leisten kann. Zum anderen aber kann man in letzter Zeit ein immer größeres Auseinanderdividieren der Qualität der Grafiksoftware feststellen. Während Profi-Mal-Programme immer schnellere und leistungsfähigere Grafikbefehle zur Verfügung stellen, tritt die veröffentlichte Literatur qualitativ auf der Stelle. Das berechtigte Informationsbedürfnis des Anwenders, wie die Steigerung der Softwarequalität zu begründen ist, konnte bislang nicht befriedigt werden. Eine Erweiterung wie Simon's Basic, die noch 1983 alle von den Stühlen riß, und über die mehrere Bücher verfaßt wurden, ist heute im Profi-Painter-Zeitalter nicht mehr salonfähig.

Ich möchte Ihnen daher an Hand von ausgewählten Beispielen zeigen, welche Möglichkeiten man bei der Grafikprogrammierung besitzt, wenn man die Sache von einer neuen, bislang nicht bekannten Seite angeht. Natürlich kann ich Ihnen keine komplette Grafikerweiterung vorstellen, da so der Rahmen dieses Buches gesprengt würde. Daher habe ich mich auf vier Routinen beschränkt, die ich allerdings in allen Einzelheiten darlegen werde:

- Setzen/Löschen/Invertieren eines Punktes
- Zeichnen eines Rechtecks
- Zeichnen eines Kreises/einer Ellipse
- Schreiben von Text in die Grafik

Mit Ausnahme des Text-Befehls wurde hauptsächlich auf die Geschwindigkeit geachtet, beim Text-Befehl stand die Anzahl der Anwendungsmöglichkeiten im Vordergrund. Fest steht, daß Sie sich auf mit die leistungsfähigsten Grafikbefehle überhaupt freuen können.

4.1 Lage der HiRes-Grafik und des Farb-RAMs

Bevor wir uns mit den einzelnen Grafik-Befehlen beschäftigen können, müssen wir zunächst einmal wissen, wie unser C64 eine Grafik aufbaut. Es ist bekannt, daß eine hochauflösende Grafik ein Punkteraster von 320 * 200 Punkten zur Verfügung stellt, insgesamt also 64000 Punkte verwalten muß. Das elementare Problem besteht darin, daß man irgendwie Informationen haben muß, ob ein Punkt gesetzt ist oder nicht. Der VIC-Chip, der für die Grafikverwaltung zuständig ist, löst das Problem dadurch, daß er jedem Punkt auf dem Bildschirm genau ein Bit des Speichers zuordnet. Wie dies genau geschieht, werden wir später besprechen. Zunächst ist nur wichtig, daß ein gesetztes Bit einem gesetzten Punkt entspricht, ein nicht gesetztes Bit daher einem gelöschten Punkt. Auf diese Weise sind 64000 Bit oder 8000 Byte erforderlich, um eine HiRes-Grafik verwalten zu können.

Als nächstes ist die Frage zu klären, welche 8000 Byte denn vom VIC als Grafikspeicher interpretiert werden, denn der C64 stellt ja insgesamt 64000 Byte an Speicherraum zur Verfügung. Nun, im Normalfall, d.h. im Textmodus, natürlich überhaupt keine. Zunächst müssen wir dem VIC mitteilen, daß wir eine Grafik darstellen wollen. Dies geschieht dadurch, daß Bit 5 des VIC-Registers 17 gesetzt wird. Erst dann wird ein bestimmter Speicherbereich als Grafikspeicher interpretiert.

Wie Sie vielleicht noch aus Kapitel 1 wissen, kann der VIC immer nur einen Bereich von 16 Kbyte = 16384 Byte adressieren. Normalerweise ist dies der Bereich von $0000 $3FFF. Leider ist es nicht so, daß wir unseren Grafikspeicher beliebig innerhalb dieses Adreßraumes plazieren können. Wir haben leider nur genau zwei Möglichkeiten: zum einen am Anfang, zum anderen ab Anfang plus $2000 des Adreßraumes. Diese Auswahl können wir durch Bit 3 des VIC-Registers 24 treffen. Ein gelöschtes Bit bedeutet, daß der Grafikspeicher in den unteren, ein gesetztes Bit, daß er in den oberen 8 Kbyte des VIC-Adreßraumes zu finden ist.

Es fällt leider sofort auf, daß wir den Raum ab $0000 überhaupt nicht benutzen können, da sich in diesem Bereich der Stack und die lebenswichtigen Vektoren des Betriebssystems befinden. Es bleibt der Bereich ab $2000, der jedoch ebenso ungünstig liegt, da wir damit Programme nur noch von $0801 bis $1FFF speichern können.

Die Lösung dieses Problems besteht darin, daß wir einfach den Adreßraum des VIC verschieben. Dies ist ja in 16-Kbyte-Schritten möglich. Die Auswahl des vom VIC adressierten Speicherbereichs wird durch die untersten beiden Bits des Registers 0 der CIA 2 (ab $DD00) getroffen. Diese beiden Bits kann man als Bit 14 und 15 der Anfangsadresse ansehen, wobei man jedoch beachten muß, daß sie low-aktiv sind, d.h., ein gesetztes Bit gilt als nicht gesetzt und umgedreht. Wenn Sie z.B. dieses Register im Normalmodus (VIC-Adreßraum von $0000-$3FFF) mit PEEK(56576) auslesen, erhalten Sie als Wert eine 3, d.h., die Bits 0 und 1 sind gesetzt, wie es bei den low-aktiven Bits sein muß.

Durch die Verlagerung des Adreßraumes kann man den Grafikspeicher an insgesamt 8 Stellen (4 Adreßbereiche * 2 Lagemöglichkeiten pro Adreßraum) positionieren. In der Praxis wird er fast immer ab $E000 unter das Betriebssystem gelegt, da er an dieser Stelle kein Programm stört. Wir wollen einmal sehen, welche Bits dafür gesetzt werden müssen: Zunächst muß der VIC-Adreßraum ganz nach oben, d.h. nach $C000 gelegt werden. Dafür müssen die Bits 0 und 1 der CIA 2 (Register 0) gelöscht werden. Zusätzlich müssen wir dem VIC mitteilen, daß sich der Grafikspeicher in den oberen 8 Kbyte befinden soll, d.h. Bit 3 des VIC-Registers 24 muß gesetzt werden. Wenn wir dann noch auf den Grafikmodus umschalten, indem wir Bit 5 des Registers 17 setzen, sind wir fertig. Zum Schluß muß noch gesagt werden, daß sich der VIC seine Informationen immer aus dem RAM holt, nicht also etwa das Betriebssystem als Grafikspeicher interpretiert!

Der nächste wichtige Punkt besteht in der Farbgebung. Wie Sie wahrscheinlich wissen, kann der VIC insgesamt 16 Farben darstellen. Für die Codierung der Farbe eines Punktes wären damit also 4 Bit erforderlich, bei 64 000 Punkten also 32 000 Byte. Dies ist natürlich nicht denkbar, vielmehr wird durch ein Byte des Farb-RAMs genau 64 Punkten eine Farbe zugeordnet. Warum dies genau 64 sind, werden wir sehen, wenn wir den Aufbau des Grafikspeichers besprechen. Damit sind 1000 Byte für die Farbgebung zuständig. Jedes Byte ist so aufgebaut, daß die untersten 4 Bit für die Hintergrundfarbe, d.h. die nicht gesetzten Punkte und die obersten 4 Bit für die gesetzten Punkte zuständig sind. Um also den Farbcode zu erhalten, muß man Punktfarbe*16 + Hintergrundfarbe rechnen.

Die Lage des Farb-RAMs innerhalb des VIC-Adreßraumes läßt sich durch die Bits 4–7 des VIC-Registers 24 festlegen. Diese fungieren als Adreßbits 10–13. Die kleinste Einheit, in der man das Farb-RAM verschieben kann, ist also 2↑10 = 1024 = 1Kbyte. Damit sind also 16 Startadressen innerhalb des Adreßbereiches möglich. Bei einer Lage des Grafikspeichers ab $E000 bietet es sich an, das Farb-RAM im Bereich von $D000 bis $DFFF abzulegen, da dieser Bereich ebenfalls nicht ohne weiteres für Programme genutzt werden kann. Um z.B. den Start nach $DC00 zu verlegen, muß man die Differenz zur Basisadresse des Adreßraumes ($C000) bilden und das Ergebnis in die Bits aufschlüsseln. Hier also die Differenz $1C00 (7168) = 2↑12 + 2↑11 + 2↑10.

Daher müssen die Bits 4, 5 und 6 des VIC-Registers 24 gesetzt werden, während das Bit 7 gelöscht werden muß, da dieses ja zum Adreßbit 13 (2↑13) korrespondiert.

Als Vorbereitung für unsere Grafikroutinen sind daher drei Dinge erforderlich: erstens Einschalten der Grafik, zweitens Setzen der Farbe, indem das Farb-RAM mit dem entsprechenden Code beschrieben wird, und drittens Löschen des Grafikspeichers, indem man ihn komplett mit Null-Bytes (= keine gesetzten Punkte) füllt. Letzteres ist erforderlich, da der Rechner nach dem Einschalten zufällige Werte produziert. Wenn man dann die Grafik einschalten würde, erschiene jedes zufällige Bit als Grafikpunkt, was natürlich unerwünscht wäre.

Hier nun diese drei Routinen, die durch eine ergänzt wurden, welche vom Grafik- in den Textmodus zurückschaltet:

Listing: »grafikroutinen«

```
0:      6000          -;routinen1:     6000          -         .ba
$6000
2:                    -;
6:      aefd          -          .eq komma     =    $aefd
7:      b79e          -          .eq byte      =    $b79e
8:                    -;
12:                   -;grafik anschalten
13:                   -;=================
14:                   -;
15:     6000 a9 00    -gron      lda  #00        ;cia adressbereich
16:     6002 8d 00 dd-           sta  $dd00      ;ab $c000 + x
17:     6005 ad 11 d0-           lda  $d011      ;bit 5 im
18:     6008 09 20    -          ora  #%00100000 ;vic-reg 17 setzen
19:     600a 8d 11 d0-           sta  $d011      ;(=grafikmodus an)
20:     600d ad 18 d0-           lda  $d018      ;bit 3 im vic-reg 24
21:     6010 09 08    -          ora  #%00001000 ;setzen  (x=$2000)
22:     6012 09 70    -          ora  #%01110000 ;videoram ab $c000
23:     6014 8d 18 d0-           sta  $d018      ;+ 7*$0400 = $dc00
24:     6017 60       -          rts
31:                   -;
32:                   -;grafik ausschalten
33:                   -;==================
34:                   -;
35:     6018 a9 c7    -groff     lda  #199       ;cia adressbereich
36:     601a 8d 00 dd-           sta  $dd00      ;ab $0000 + x
37:     601d ad 11 d0-           lda  $d011      ;bit 5 im
38:     6020 29 df    -          and  #%11011111 ;vic-reg17 loeschen
39:     6022 8d 11 d0-           sta  $d011      ;(=grafikmodus aus)
40:     6025 ad 18 d0-           lda  $d018      ;bit3 im vic-reg 24
41:     6028 29 f7    -          and  #%11110111 ;loeschen (x=$0000)
42:     602a 29 1f    -          and  #%00011111 ;video-ram ab $0000
43:     602c 8d 18 d0-           sta  $d018      ;+ 1*$0400 = $0400
44:     602f 60       -          rts
51:                   -;
52:                   -;farbe setzen
53:                   -;============
54:                   -;
55:     6030 20 fd ae-color      jsr  komma
56:     6033 20 9e b7-           jsr  byte       ;punktfarbe holen
57:     6036 e0 10    -          cpx  #16        ;= 16, dann fehler
58:     6038 b0 31    -          bcs  error
59:     603a 8a       -          txa
60:     603b 0a       -          asl  a
61:     603c 0a       -          asl  a
62:     603d 0a       -          asl  a          ; mal 16
```

```
603e 0a       -            asl  a
603f 85 fa    -            sta  $fa         ;merken
6041 20 fd ae-             jsr  komma       ;hintergrundfarbe
6044 20 9e b7-             jsr  byte        ;holen
6047 e0 10    -            cpx  #16         ;=16, dann fehler
6049 b0 20    -            bcs  error
604b 8a       -            txa
604c 18       -            clc
604d 65 fa    -            adc  $fa         ;plus punktfarbe*16
604f a0 00    -            ldy  #$00
6051 78       -            sei              ;auf ram schalten
6052 a2 34    -            ldx  #52
6054 86 01    -            stx  $01
6056 99 ff db-lf           sta  $dbff,y     ;wert ins
6059 99 ff dc-             sta  $dcff,y
605c 99 ff dd-             sta  $ddff,y     ;farbram schreiben
605f 99 ff de-             sta  $deff,y
6062 88       -            dey
6063 d0 f1    -            bne  lf
6065 a2 37    -            ldx  #55         ;auf rom schalten
6067 86 01    -            stx  $01
6069 58       -            cli
606a 60       -            rts
              -;
606b 4c 48 b2-error        jmp  $b248       ;illegal quantity
              -;
              -;bildschirm loeschen
              -;==================
              -;
606e a9 00    -clear       lda  #00         ;loeschwert
6070 a2 00    -            ldx  #e000
6072 86 fa    -            stx  $fa
6074 a2 e0    -            ldx  #$e000
6076 86 fb    -            stx  $fb
6078 a2 20    -            ldx  #32         ;32 pages loeschen
607a a0 00    -c1          ldy  #00
607c 91 fa    -c2          sta  ($fa),y
607e c8       -            iny
607f d0 fb    -            bne  c2          ;ins grafik-ram
6081 e6 fb    -            inc  $fb
6083 ca       -            dex              ;schreiben
6084 d0 f4    -            bne  c1
6086 60       -            rts
              -;
```

Sie können die einzelnen Routinen folgendermaßen aufrufen:

Grafik einschalten:	SYS 24576
Grafik ausschalten:	SYS 24600
Farbe setzen:	SYS 24624, PF, HF
Grafik löschen:	SYS 24686

Mit PF wurde die Punktfarbe, mit HF die Hintergrundfarbe bezeichnet (Farbe der nicht gesetzten Punkte).

Falls Sie den Grafikspeicher und/oder das Farb-RAM woanders hinlegen möchten, habe ich in zwei Tabellen den Zusammenhang zwischen den entsprechenden Bits und der jeweiligen Lage dargestellt, zunächst der Grafikspeicher:

CIA R 0 Bit 0	CIA R 0 Bit 1	VIC R24 Bit 3	Startadresse Speicher
1	1	0	0+ 0+ 0= 0
1	1	1	0+ 0+ 8192= 8192
1	0	0	0+ 16384+ 0= 16384
1	0	1	0+ 16384+ 8192= 24576
0	1	0	32768+ 0+ 0= 32768
0	1	1	32768+ 0+ 8192= 40960
0	0	0	32768+ 16384+ 0= 49152
0	0	1	32768+ 16384+ 8192= 57344

Nun zum Farb-RAM:

V I Bit 7	C-R Bit 7	E G. Bit 5	24 Bit 4	Startadresse Farb-RAM
0	0	0	0	VIC-Adreßbereich + 0
0	0	0	1	VIC-Adreßbereich + 1024
0	0	1	0	VIC-Adreßbereich + 2048
0	0	1	1	VIC-Adreßbereich + 3072
0	1	0	0	VIC-Adreßbereich + 4096
0	1	0	1	VIC-Adreßbereich + 5120
0	1	1	0	VIC-Adreßbereich + 6144

V I Bit 7	C-R Bit 7	E G. Bit 5	24 Bit 4	Startadresse Farb-RAM
0	1	1	1	VIC-Adreßbereich + 7168
1	0	0	0	VIC-Adreßbereich + 8192
1	0	0	1	VIC-Adreßbereich + 9216
1	0	1	0	VIC-Adreßbereich + 10240
1	0	1	1	VIC-Adreßbereich + 11264
1	1	0	0	VIC-Adreßbereich + 12288
1	1	0	1	VIC-Adreßbereich + 13312
1	1	1	0	VIC-Adreßbereich + 14336
1	1	1	1	VIC-Adreßbereich + 15360

Nachdem wir nun über die Lage des Grafikspeichers und des Farb-RAMs Bescheid wissen, kommen wir zur nächsten wichtigen Frage: Wie sind die beiden Speicher aufgebaut, d.h. welches Bit ist für welchen Punkt zuständig ? Diese Fragen werden wir im nächsten Kapitel klären.

4.2 Aufbau des Grafikspeichers und des Farb-RAMs

Wie schon gesagt wurde, besitzt eine HiRes-Grafik eine Auflösung von 320 Punkten (Bits) in X-Richtung sowie von 200 Punkten (Bits) in Y-Richtung. Die Frage ist jetzt, wie man einen Zusammenhang von Bits und Punktkoordinaten herstellt. Wenn man sich den Grafikschirm bildlich vorstellt, beginnt der Grafikspeicher in der linken oberen Ecke. Das erste Byte stellt 8 Punkte in X-Richtung dar, wobei das höherwertigste Bit am weitesten links liegt, beim ersten Byte des Speichers überhaupt also ganz links oben in der Ecke. Man könnte nun annehmen, daß 40 Byte (=320 Bit) direkt aneinander liegen würden, so daß die ersten 40 Byte des Grafikspeichers die oberste Zeile des Grafikschirms darstellen würden. Dies ist jedoch nicht der Fall. Der Aufbau entspricht vielmehr dem des Zeichengenerators, d.h. immer 8 Byte liegen blockweise untereinander, bevor es in X-Richtung weitergeht. Bildlich gesprochen heißt dies, daß die ersten 8 Byte des Grafikspeichers die Zeilen 1–8 jeweils 8 Punkte (Bit) breit darstellen. Wenn auf diese Weise 320 Spalten und 8 Zeilen überdeckt wurden, geht das gleiche von vorn los, jedoch um 8 Zeilen nach unten versetzt. Da der Aufbau sprachlich nur sehr schwer zu schildern ist, sehen Sie auf der nächsten Seite eine ausschnittsweise Zeichnung des Grafikschirms, wobei die Startadresse des Grafikspeichers bei 57 344 liegen soll.

Der Vorteil dieser Anordnung liegt darin, daß der Aufbau des Farb-RAMs um so einfacher ist. Im vorigen Kapitel sagte ich, daß ein Byte des Farb-RAMs für 64 Punkte zuständig ist.

Spalte	00000000	00111111	11112222	22222233	....	33333333
	01234567	89012345	67890123	45678901	....	11111111
						23456789
Bits	76543210	76543210	76543210	76543210	....	76543210
Zeile 00	57344	57352	57360	57368	....	57656
01	57345	57353	57361	57369	....	57657
02	57346	57354	57362	57370	....	57658
03	57347	57355	57363	57371	....	57659
04	57348	57356	57364	57372	....	57660
05	57349	57357	57365	57373	....	57661
06	57350	57358	57366	57374	....	57662
07	57351	57359	57367	57375	....	57663
08	57664	57672	57680	57688	....	57976
09	57665	57673	57681	57689	....	57977
10	57666	57674	57682	57690	....	57978
11	57667	57675	57683	57691	....	57979
12	57668	57676	57684	57692	....	57980
13	57669	57677	57685	57693	....	57981
14	57670	57678	57686	57694	....	57982
15	57671	57679	57687	57695	....	57983
	.	.	.	.	.	.
	.	.	.	.	.	.
184	64704	64712	64720	64728	....	65016
185	57345	57353	57361	57369	....	57657
186	57346	57354	57362	57370	....	57658
187	57347	57355	57363	57371	....	57659
188	57348	57356	57364	57372	....	57660
189	57349	57357	57365	57373	....	57661
190	57350	57358	57366	57374	....	57662
191	57351	57359	57367	57375	....	57663
192	65024	65032	65040	65048	....	65336
193	65025	65033	65041	65049	....	65337
194	65026	65034	65042	65050	....	65338
195	65027	65035	65043	65051	....	65339
196	65028	65036	65044	65052	....	65340
197	65029	65037	65045	65053	....	65341
198	65030	65038	65046	65054	....	65342
199	65031	65039	65047	65055	....	65343

Bild 4.1.: *Aufbau des Grafikspeichers ab 57344 ($E000) (Li. ob. Ecke = Spalte 0, Zeile 0)*

Spalte	00000000 01234567	00111111 89012345	11112222 67890123	22222233 45678901	....	33333333 11111111 23456789
Zeile 00	56320	56321	56322	56323	....	56359
01	56320	56321	56322	56323	....	56359
02	56320	56321	56322	56323	....	56359
03	56320	56321	56322	56323	....	56359
04	56320	56321	56322	56323	....	56359
05	56320	56321	56322	56323	....	56359
06	56320	56321	56322	56323	....	56359
07	56320	56321	56322	56323	....	56359
08	56360	56361	56362	56363	....	56399
09	56360	56361	56362	56363	....	56399
10	56360	56361	56362	56363	....	56399
11	56360	56361	56362	56363	....	56399
12	56360	56361	56362	56363	....	56399
13	56360	56361	56362	56363	....	56399
14	56360	56361	56362	56363	....	56399
15	56360	56361	56362	56363	....	56399
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
184	57240	57241	57242	57243	....	57279
185	57240	57241	57242	57243	....	57279
186	57240	57241	57242	57243	....	57279
187	57240	57241	57242	57243	....	57279
188	57240	57241	57242	57243	....	57279
189	57240	57241	57242	57243	....	57279
190	57240	57241	57242	57243	....	57279
191	57240	57241	57242	57243	....	57279
192	57280	57281	57282	57283	....	57319
193	57280	57281	57282	57283	....	57319
194	57280	57281	57282	57283	....	57319
195	57280	57281	57282	57283	....	57319
196	57280	57281	57282	57283	....	57319
197	57280	57281	57282	57283	....	57319
198	57280	57281	57282	57283	....	57319

Bild 4.2.: *Aufbau des Farb-RAMs ab 56320 ($DC00)*

Es liegt praktisch auf der Hand, jedem 8*8-Punkte-Block ein solches Byte zuzuordnen. Damit entspricht der Aufbau des Farb-RAMs genau dem des Textbildschirms, in dem die Bytes von links nach rechts und von oben nach unten aufgebaut werden. Auch hierzu befindet sich eine Zeichnung auf der übernächsten Seite.

Das Grundproblem für den Anwender besteht nun darin, einen Algorithmus zu entwerfen, der es erlaubt, den einzelnen Punkten Koordinaten zu geben und abhängig von diesen ein bestimmtes Bit des Grafikspeichers anzusprechen. Es ist ja für den Menschen kaum möglich, in Bits und Bytes zu denken, vielmehr ist er es gewohnt, in x- und y-Koordinaten zu rechnen und auch zu denken.

Am Anfang muß man sich überlegen, wohin man den Koordinatenursprung legt. Man könnte diesen z.B. genau in die Mitte des Bildschirms legen, hätte dann aber den Nachteil, in positiven und negativen Koordinaten rechnen zu müssen. Um dies zu vermeiden, ist es üblich geworden, ihn in die linke obere Ecke des Grafikschirms zu legen und von dort aus die X-Koordinate nach rechts und die Y-Koordinate nach unten zu rechnen. Die Berechnung der zuständigen Adresse für ein Koordinatenpaar ist nicht so schwierig wie es zunächst scheint:

Zunächst wollen wir den X-Offset berechnen. Wenn die Bytes nacheinander in einer Zeile liegen würden, könnte man das entsprechende Byte durch die Rechnung INT(X/8) berechnen, da jeweils die x-Koordinaten 0–7, 8–15 usw. in einem Byte liegen. Da es sich jedoch um Achter-Blöcke handelt (siehe Bild 4.1.), müssen wir das Ergebnis nochmals mit 8 multiplizieren: XOFF=INT(X/8)*8.

In Maschinensprache bedeutet dies, daß man das entsprechende Byte dreimal um ein Bit nach rechts rotieren läßt (=/2/2/2=/8) und anschließend wieder um drei Bit nach links (=*2*2*2=*8). Die untersten drei Bit sind dabei verlorengegangen, was der INT-Funktion entspricht. Einfacher kann man diesen Effekt erreichen, indem man die untersten drei Bit einfach mit dem Befehl AND %11111000 ausblendet.

In Y-Richtung muß man zwischen zwei Fällen unterscheiden: Falls die Koordinate kleiner als 8 ist, kann man sie einfach hinzuaddieren. Falls sie jedoch größer oder gleich 8 ist, muß man pro Zeile 40 Byte hinzuaddieren (siehe Bild 4.1.), pro Block 320 Byte, so daß man rechnen kann: YOFF=40*INT(Y/8)*8. Auch hier kommt natürlich in der Praxis der AND %11111000-Befehl zum Einsatz. Zu diesem Offset kann man dann den »Rest« der Y-Koordinate hinzuaddieren, was durch den Befehl AND #07 erreicht werden kann, da dieser nur die untersten drei Bit berücksichtigt. Die endgültige Adresse erhält man also durch folgende Formel:

Adresse = Start Grafikspeicher + XAND248 + 40*YAND248 + YAND7

Wir wollen diese Rechnung nun in Maschinensprache nachvollziehen, wobei wir die X-Koordinaten jedoch in zwei Byte aufteilen müssen, da sie größer als 255 werden und damit nicht mehr in einem Byte untergebracht werden können. Wir wollen das Lowbyte mit »xl«, das Highbyte mit »xh« und die y-Koordinate mit »y« bezeichnen. Mit R1 und R2 werden zwei Speicherstellen bezeichnet, die zum Rechnen benötigt werden.

Zunächst wollen wir den Y-Term abhandeln:

```
LDA Y          ; Y-Koordinate holen
AND #$F8       ; INT(Y/8)*8 bilden
PHA            ; merken
```

Nun müssen wir diesen Ausdruck mal 40 nehmen. Dies ist unmittelbar nicht möglich, daher unterteilen wir die Multiplikation in 3 Schritte:

```
A*40 = (A*2*2 + A)*2*2*2, da 40=8*5
     STA R1         ; A=(INT(Y/8)*8)*2*2
     LDA #00        ; berechnen
     STA R2
     ASL R1
     ROL R2
     ASL R1
     ROL R2
     PLA            ; A=A+INT(Y/8)*8
     CLC            ; berechnen
     ADC R1
     STA R1
     BCC WEI1
     INC R2
WEI1 ASL R1         ; A=A*2*2*2
     ROL R2         ; berechnen
     ASL R1
     ROL R2
     ASL R1
     ROL R2
```

Als nächstes können wir den Term YAND7 berechnen und hinzuaddieren, natürlich ist auch dies eine 16-Bit-Operation:

```
     LDA Y
     AND #07
     CLC
     ADC R1
     STA R1
     BCC WEI2
     INC R2
WEI2 NOP
```

Nun müssen wir als letztes noch den Wert XAND248 berücksichtigen und anschließend noch die Startadresse unseres Grafikspeichers hinzuaddieren, um anschließend die Adresse des Grafikbytes in R1/R2 vorliegen zu haben:

```
LDA XL
AND #$F8
CLC
ADC R1
STA R1
LDA XH
ADC R2
ADC #$E0      ; Highbyte von $E000
STA R2
```

Zum Schluß müssen wir nur noch die Bitposition innerhalb des Bytes bestimmen. Hierfür sind nur die untersten drei Bits relevant, wobei sich eine reziproke Zuordnung ergibt:

X Bitposition
0 – 7
1 – 6
2 – 5
3 – 4
4 –3
5 – 2
6 – 1
7 – 0

Mit diesem Wert muß die Zweierpotenz gebildet werden, also insgesamt Bit=2↑(7-XAND7). Da dies umständlich zu berechnen ist, helfen wir uns mit einer Tabelle aus, in der die Potenzen gespeichert sind:

```
LDA XL
AND #07
TAX
LDA TAB,X

TAB  .Byte 128,64,32,16,8,4,2,1
```

Wollen wir einen Punkt setzen, müssen wir den obigen Wert in das Byte einblenden. Hierzu können wir die ORA-Funktion verwenden:

```
LDY #00
ORA (R1),Y
STA (R1),Y
```

Es ist noch zu beachten, daß vor Aufruf der Routine der Prozessorport auf RAM geschaltet werden muß, da sonst nicht die Grafik-, sondern die Betriebssystemwerte ausgelesen werden.

Wenn Sie sich diese Routine genau ansehen, erkennen Sie, daß sie relativ viel Zeit für die Berechnung der Grafikadresse benötigt. Ich möchte Ihnen nun ein Programm vorstellen, das zu den schnellsten überhaupt gehört, jedoch wesentlich mehr Speicherplatz benötigt als die obige Routine:

Listing: »plot«

```
6088            -;plot
6088            -          .ba $6088
                -;
aefd            -          .eq komma    =     $aefd
ad9e            -          .eq frmevl   =     $ad9e
b1bf            -          .eq integer  =     $b1bf
b79e            -          .eq byte     =     $b79e
00fa            -          .eq xl       =     $fa
00fb            -          .eq xh       =     $fb
00fc            -          .eq y        =     $fc
00f9            -          .eq modus    =     $f9
                -;
                -;punkt setzen/loeschen/invertieren
                -;=================================
                -;
6088 a6 fc      -plot1     ldx  y        ;y holen
608a e0 c8      -          cpx  #200     ;=200
608c b0 2f      -          bcs  out      ;ja, dann zurueck
608e a4 fa      -          ldy  xl       ;x-low holen
6090 a5 fb      -          lda  xh       ;x-high holen
6092 f0 08      -          beq  p1       ;0, dann xl beliebig
6094 c9 02      -          cmp  #02      ;=2
6096 b0 25      -          bcs  out      ;ja, dann zurueck
6098 c0 40      -          cpy  #64      ;xh =1 und xl =64
609a b0 21      -          bcs  out      ;ja, dann zurueck
609c 98         -p1        tya
609d 29 f8      -          and  #$f8     ;(int(x/8))*8
609f 18         -          clc
60a0 7d c8 60-             adc  tabl,x   ;+anfangsadresse
60a3 85 9e      -          sta  $9e      ;der durch y
60a5 a5 fb      -          lda  xh       ;festgelegten
60a7 7d 90 61-             adc  tabh,x   ;grafikzeile
60aa 85 9f      -          sta  $9f      ;=grafikbyte
60ac b9 58 62-             lda  bits,y   ;2↑(int(x/8))=bit
60af a0 00      -          ldy  #00      ;im grafikbyte
60b1 24 f9      -          bit  modus    ;modus-byte pruefen
60b3 70 09      -          bvs  setz     ;bit 5=1, setzen
60b5 30 0c      -          bmi  inv      ;bit 7=1,invertieren
                -;
                -;punkt loeschen
```

```
                 -;==============
                 -;
60b7 49 ff       -loesch     eor   #$ff
60b9 31 9e       -           and   ($9e),y ;bit ausblenden
60bb 91 9e       -           sta   ($9e),y
60bd 60          -out        rts
                 -;
                 -;punkt setzen
                 -;============
                 -;
60be 11 9e       -setz       ora   ($9e),y
60c0 91 9e       -           sta   ($9e),y ;bit einblenden
60c2 60          -           rts
                 -;
                 -;punkt invertieren
                 -;=================
60c3 51 9e       -inv        eor   ($9e),y
60c5 91 9e       -           sta   ($9e),y ;bit umdrehen
60c7 60          -           rts
                 -;
60c8 00 01 02-tabl           .by$00,$01,$02,$03,$04,$05
60ce 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
60d5 45 46 47-               .by$45,$46,$47,$80,$81,$82,$83
60dc 84 85 86-               .by$84,$85,$86,$87,$c0,$c1,$c2
60e3 c3 c4 c5-               .by$c3,$c4,$c5,$c6,$c7
60e8 00 01 02-               .by$00,$01,$02,$03,$04,$05
60ee 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
60f5 45 46 47-               .by$45,$46,$47,$80,$81,$82,$83
60fc 84 85 86-               .by$84,$85,$86,$87,$c0,$c1,$c2
6103 c3 c4 c5-               .by$c3,$c4,$c5,$c6,$c7
6108 00 01 02-               .by$00,$01,$02,$03,$04,$05
610e 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
6115 45 46 47-               .by$45,$46,$47,$80,$81,$82,$83
611c 84 85 86-               .by$84,$85,$86,$87,$c0,$c1,$c2
6123 c3 c4 c5-               .by$c3,$c4,$c5,$c6,$c7
6128 00 01 02-               .by$00,$01,$02,$03,$04,$05
612e 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
6135 45 46 47-               .by$45,$46,$47,$80,$81,$82,$83
613c 84 85 86-               .by$84,$85,$86,$87,$c0,$c1,$c2
6143 c3 c4 c5-               .by$c3,$c4,$c5,$c6,$c7
6148 00 01 02-               .by$00,$01,$02,$03,$04,$05
614e 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
6155 45 46 47-               .by$45,$46,$47,$80,$81,$82,$83
615c 84 85 86-               .by$84,$85,$86,$87,$c0,$c1,$c2
6163 c3 c4 c5-               .by$c3,$c4,$c5,$c6,$c7
6168 00 01 02-               .by$00,$01,$02,$03,$04,$05
616e 06 07 40-               .by$06,$07,$40,$41,$42,$43,$44
```

```
94:    6175 45 46 47-        .by$45,$46,$47,$80,$81,$82,$83
95:    617c 84 85 86-        .by$84,$85,$86,$87,$c0,$c1,$c2
96:    6183 c3 c4 c6-        .by$c3,$c4,$c5,$c6,$c7
97:    6188 00 01 02-        .by$00,$01,$02,$03,$04,$05
98:    618e 06 07   -        .by$06,$07
99:                 -;
100:   6190 e0 e0 e0-tabh    .by$e0,$e0,$e0,$e0,$e0,$e0
101:   6196 e0 e0 e1-        .by$e0,$e0,$e1,$e1,$e1,$e1,$e1
102:   619d e1 e1 e1-        .by$e1,$e1,$e1,$e2,$e2,$e2,$e2
103:   61a4 e2 e2 e2-        .by$e2,$e2,$e2,$e2,$e3,$e3,$e3
104:   61ab e3 e3 e3-        .by$e3,$e3,$e3,$e3,$e3
105:   61b0 e5 e5 e5-        .by$e5,$e5,$e5,$e5,$e5,$e5
106:   61b6 e5 e5 e6-        .by$e5,$e5,$e6,$e6,$e6,$e6,$e6
107:   61bd e6 e6 e6-        .by$e6,$e6,$e6,$e7,$e7,$e7,$e7
108:   61c4 e7 e7 e7-        .by$e7,$e7,$e7,$e7,$e8,$e8,$e8
109:   61cb e8 e8 e8-        .by$e8,$e8,$e8,$e8,$e8
110:   61d0 ea ea ea-        .by$ea,$ea,$ea,$ea,$ea,$ea
111:   61d6 ea ea eb-        .by$ea,$ea,$eb,$eb,$eb,$eb,$eb
112:   61dd eb eb eb-        .by$eb,$eb,$eb,$ec,$ec,$ec,$ec
113:   61e4 ec ec ec-        .by$ec,$ec,$ec,$ec,$ed,$ed,$ed
114:   61eb ed ed ed-        .by$ed,$ed,$ed,$ed,$ed
115:   61f0 ef ef ef-        .by$ef,$ef,$ef,$ef,$ef,$ef
116:   61f6 ef ef f0-        .by$ef,$ef,$f0,$f0,$f0,$f0,$f0
117:   61fd f0 f0 f0-        .by$f0,$f0,$f0,$f1,$f1,$f1,$f1
118:   6204 f1 f1 f1-        .by$f1,$f1,$f1,$f1,$f2,$f2,$f2
119:   620b f2 f2 f2-        .by$f2,$f2,$f2,$f2,$f2
120:   6210 f4 f4 f4-        .by$f4,$f4,$f4,$f4,$f4,$f4
121:   6216 f4 f4 f5-        .by$f4,$f4,$f5,$f5,$f5,$f5,$f5
122:   621d f5 f5 f5-        .by$f5,$f5,$f5,$f6,$f6,$f6,$f6
123:   6224 f6 f6 f6-        .by$f6,$f6,$f6,$f6,$f7,$f7,$f7
124:   622b f7 f7 f7-        .by$f7,$f7,$f7,$f7,$f7
125:   6230 f9 f9 f9-        .by$f9,$f9,$f9,$f9,$f9,$f9
126:   6236 f9 f9 fa-        .by$f9,$f9,$fa,$fa,$fa,$fa,$fa
127:   623d fa fa fa-        .by$fa,$fa,$fa,$fb,$fb,$fb,$fb
128:   6244 fb fb fb-        .by$fb,$fb,$fb,$fb,$fc,$fc,$fc
129:   624b fc fc fc-        .by$fc,$fc,$fc,$fc,$fc
130:   6250 fe fe fe-        .by$fe,$fe,$fe,$fe,$fe,$fe
131:   6256 fe fe   -        .by$fe,$fe
140:                 -;
141:   6258 80 40 20-bits    .by128,64,32,16,8,4,2,1
142:   6260 80 40 20-        .by128,64,32,16,8,4,2,1
143:   6268 80 40 20-        .by128,64,32,16,8,4,2,1
144:   6270 80 40 20-        .by128,64,32,16,8,4,2,1
145:   6278 80 40 20-        .by128,64,32,16,8,4,2,1
146:   6280 80 40 20-        .by128,64,32,16,8,4,2,1
147:   6288 80 40 20-        .by128,64,32,16,8,4,2,1
148:   6290 80 40 20-        .by128,64,32,16,8,4,2,1
```

```
6298 80 40 20-          .by128,64,32,16,8,4,2,1
62a0 80 40 20-          .by128,64,32,16,8,4,2,1
62a8 80 40 20-          .by128,64,32,16,8,4,2,1
62b0 80 40 20-          .by128,64,32,16,8,4,2,1
62b8 80 40 20-          .by128,64,32,16,8,4,2,1
62c0 80 40 20-          .by128,64,32,16,8,4,2,1
62c8 80 40 20-          .by128,64,32,16,8,4,2,1
62d0 80 40 20-          .by128,64,32,16,8,4,2,1
62d8 80 40 20-          .by128,64,32,16,8,4,2,1
62e0 80 40 20-          .by128,64,32,16,8,4,2,1
62e8 80 40 20-          .by128,64,32,16,8,4,2,1
62f0 80 40 20-          .by128,64,32,16,8,4,2,1
62f8 80 40 20-          .by128,64,32,16,8,4,2,1
6300 80 40 20-          .by128,64,32,16,8,4,2,1
6308 80 40 20-          .by128,64,32,16,8,4,2,1
6310 80 40 20-          .by128,64,32,16,8,4,2,1
6318 80 40 20-          .by128,64,32,16,8,4,2,1
6320 80 40 20-          .by128,64,32,16,8,4,2,1
6328 80 40 20-          .by128,64,32,16,8,4,2,1
6330 80 40 20-          .by128,64,32,16,8,4,2,1
6338 80 40 20-          .by128,64,32,16,8,4,2,1
6340 80 40 20-          .by128,64,32,16,8,4,2,1
6348 80 40 20-          .by128,64,32,16,8,4,2,1
6350 80 40 20-          .by128,64,32,16,8,4,2,1
             -;
             -;koordinaten holen
             -;=================
             -;
6358 20 9e ad-koor      jsr  frmevl  ;ausdruck holen
635b 20 bf b1-          jsr  integer ;x nach integer
635e a5 65   -          lda  $65
6360 85 fa   -          sta  xl      ;x-low
6362 a5 64   -          lda  $64
6364 85 fb   -          sta  xh      ;x-high
6366 20 fd ae-          jsr  komma
6369 20 9e b7-          jsr  byte    ;byte holen
636c 86 fc   -          stx  y       ;y-koordinate
636e 20 fd ae-          jsr  komma
6371 20 9e b7-          jsr  byte    ;byte holen
6374 86 f9   -          stx  modus   ;modus
6376 60      -          rts
             -;
6377 20 fd ae-plot      jsr  komma
637a 20 58 63-          jsr  koor    ;koordinaten
637d 78      -          sei          ;holen
637e a9 34   -          lda  #52     ;auf ram
6380 85 01   -          sta  $01     ;schalten
```

```
265:    6382 20 88 60-         jsr  plotl    ;punkten
266:    6385 a9 37   -         lda  #55      ;auf rom
267:    6387 85 01   -         sta  $01      ;schalten
268:    6389 58      -         cli
269:    638a 60      -         rts
270:                 -;
```

Wie funktioniert nun diese Plot-Routine? Zunächst wird in den Zeilen 16–25 überprüft, ob die Koordinaten in dem zulässigen Bereich (»x« von 0 bis 319, »y« von 0 bis 199) liegen. Die obige umständliche Berechnung des y-Offset entfällt. Dafür existiert eine Tabelle, die für alle Y-Koordinaten die Anfangsadressen der Zeilenanfänge enthält. Zu dieser wird noch der X-Offset addiert (Zeilen 27–33). Um selbst bei der Bitmuster-Berechnung noch Zeit zu sparen, wurde diese ebenfalls abgeändert. Die Tabelle mit den Zweierpotenzen wurde auf alle X-Werte ausgedehnt, so daß die Befehle AND #07 und TAX (s.o.) entfallen konnten. Da die Zweierpotenz für z.B. x=9 dieselbe ist wie für x=2, wiederholen sich die Tabellenwerte im Abstand von 8 Byte. Man muß natürlich die Frage stellen, ob es sich lohnt, wegen der Einsparung von wenigen Taktzyklen den Umfang einer Tabelle derart zu erhöhen. Da die Plot-Routine jedoch die Grundlage schlechthin ist und von allen anderen Grafikbefehlen angesprungen wird, ist diese »Aufblähung« zugunsten der Zeit gerechtfertigt. Man muß sich darüber im klaren sein, daß die besten Kreis- oder Linienalgorithmen wertlos sind, wenn sie eine langsam arbeitende Plot-Routine anspringen, die die gewonnene Rechenzeit wieder verschlingt. Fest steht, daß die obige Routine zu den schnellsten jemals entwickelten Programmen gehört.

Wie aus dem Assemblerlisting hervorgeht, ist sie neben dem Setzen von Punkten auch in der Lage, diese zu löschen oder zu invertieren. Beim Löschen (Zeilen 46–49) wird zunächst durch den EOR-Befehl bewirkt, daß das angesprochene Bit gelöscht wird, während alle anderen Bits gesetzt werden. Danach kann man das Null-Bit durch den AND-Befehl in das Grafik-Byte einblenden. Beim Invertieren (Zeilen 60–63) wird das Bit durch den EOR-Befehl wechselweise gesetzt und gelöscht. Um eine möglichst schnelle Prüfung des Zeichenmodus zu erreichen, wird dieser nicht wie bei anderen Erweiterungen mit den Kennziffern 1, 2 oder 3 bezeichnet, sondern mit 32 (Löschen), 64 (Setzen) und 128 (Invertieren). Dadurch kann man sehr schnell mit Hilfe des BIT-Befehls in Zeile 39 den Modus überprüfen.

Aufgerufen wird der Plot-Befehl mit SYS 25463,X,Y,Modus. Durch die Basic-Routinen, welche die Parameter hinter dem SYS-Befehl aus dem Basic-Text holen, wird die Geschwindigkeit stark herabgesetzt. Erst bei den folgenden Befehlen, die dann die Plot-Routine direkt anspringen, wird deutlich, wie schnell diese Routine arbeiten kann.

4.3 Zeichnen von Rechtecken

Das Erstellen von Rechtecken ist eine typische Schwachstelle von fast allen Grafikerweiterungen. Der Grund hierfür ist relativ einleuchtend: Da in diesen Erweiterungen ein universeller Algorithmus zum Zeichnen einer Linie vorhanden ist, wird dieser einfach viermal hintereinander aufgerufen, fertig ist das Rechteck. Da ein Linienalgorithmus jedoch relativ umständlich arbeitet, sind die Ausführungszeiten nicht gerade als schnell zu bezeichnen. Außerdem hängt sie sehr stark von der Anzahl der zu setzenden Punkte (= Größe des Rechtecks) ab, was sehr unangenehm für den Benutzer ist. Um eine wesentlich schnellere Routine zu entwerfen, muß man sich nur klarmachen, daß man nur zwei Arten von Linien zu betrachten hat, nämlich jeweils zwei waagerechte und zwei senkrechte Linien. Die Berechnung dieser ist wesentlich einfacher und schneller durchzuführen, als es ein universelles Linienprogramm könnte, da hiermit ja auch schräge Linien berechnet werden müssen.

Zunächst zu den waagerechten Linien: Wir wollen als Beispiel eine Linie von X=6 nach X=42 ziehen, wobei die Y-Koordinate beliebig sein soll. Wenn man sich die Linie grafisch im Byte-Muster vorstellt, erkennt man, daß praktisch nur drei verschiedene Inhalte der Grafikbytes vorkommen: Ein Bitmuster am linken Rand, eins am rechten und dazwischen nur $FF-Bytes, da alle Bits dieser Bytes gesetzt sind.

X	00000000	00111111	11112222	22222233	33333333	444444444
	01234567	89012345	67890123	45678901	23456789	012345678
ADRESSE	Start	Start+8	Start+16	Start+24	Start+32	Start+40
Bits	01234567	01234567	01234567	01234567	01234567	01234567
Linie	—	———	———	———	———	——

Ähnlich wie bei der Plot-Routine kann man nun für den linken und rechten Rand Tabellen mit Werten in Abhängigkeit der X-Koordinate bzw. ihrer unteren drei Bit anlegen, so daß man auch die Randbytes in einem Zug erschlagen kann und nicht Punkt für Punkt setzen muß. Der Unterschied zur Bit-Tabelle der Plot-Routine besteht darin, daß nicht nur ein Bit, sondern auch alle niederwertigeren beim linken Rand bzw. alle höherwertigeren beim rechten Rand gesetzt werden müssen, wie folgende Grafik verdeutlicht:

	Linker Rand		Rechter Rand	
XAND7	76543210	Wert	76543210	Wert
0	--------	255	-	128
1	-------	127	--	192
2	------	63	---	224
3	-----	31	----	240
4	----	15	-----	248
5	---	7	------	252
6	--	3	-------	254
7	-	1	--------	255

Die Anzahl der $FF-Zwischenbytes erhält man, indem man die Differenz der Randadressen durch 8 teilt, da ja immer 8 Byte blockweise untereinanderliegen. Der Abstand von zwei Byte einer waagerechten Linie beträgt daher auch acht Byte. Um eine solche Linie zu zeichnen, braucht man also nur die Werte für den linken und rechten Rand aus der Tabelle auslesen und eine bestimmte Anzahl von $FF-Bytes setzen. Falls der rechte und linke Rand in einem Grafikbyte zu finden sind, verknüpft man einfach die beiden Tabellenwerte mit einem AND-Befehl, wodurch nur die Bits gesetzt werden, die in beiden Werten gesetzt sind:

```
    76543210
        ____
     AND
     _____
=       —
```

Bei vertikalen Linien ist die Sache fast noch einfacher: Da die X-Koordinate konstant bleibt, braucht überhaupt nur ein Bitmuster betrachtet zu werden. Das einzige Problem besteht darin, zu erkennen, ob der Abstand zum nächsten Byte genau ein Byte oder 320 Byte beträgt, wann also der Sprung von einem Block in den nächsten erfolgt, wie unser Beispiel einer Linie von Y= 4 nach Y=13 zeigt:

```
Zeile  00
   +1  01
   +2  02
   +3  03
   +4  04 ↑
   +5  05 ↑
   +6  06 ↑
   +7  07 ↑

+320   08 ↑
+321   09 ↑
+322   10 ↑
+323   11 ↑
+324   12 ↑
+325   13 ↑
+326   14
+327   15
```

Die Anzahl der Bytes errechnet sich einfach als Y2–Y1+1. Dann muß man die Position innerhalb eines Achter-Blocks berechnen, was wieder mit dem Befehl YAND7 geschehen kann. Nun bestimmt man die Startadresse des Blocks, indem man den ersten Y-Wert mit dem Befehl YAND248 ermittelt und dann wie beim Plot-Befehl die Adresse aus den Tabellen abliest. Jetzt kann man den vorher berechneten Wert der Position innerhalb des Blocks als

Index nehmen und so lange hochzählen, bis der Wert acht erreicht wurde. Dann nämlich ist ein Blockwechsel fällig und man muß die Adresse um 320 erhöhen. Mit dem Index Null geht es dann weiter, und das Spiel wiederholt sich so lange, bis alle Bytes gesetzt wurden.

Sie werden einsehen, daß eine herkömmliche Rechteck-Routine keine Chance hat, mit dieser Geschwindigkeit mitzuhalten. Die Zeichengeschwindigkeit ist praktisch unabhängig von der Größe des Rechtecks. Hier nun die ausführlich kommentierte Routine, die jeweils zwei Linien parallel zeichnet:

Listing: »rechteck«

```
100:    685d             -;rechteck
101:    685d             -          .ba $685d
103:                     -;
104:    aefd             -          .eq komma     =  $aefd ;komma,
105:    ad8a             -          .eq frmnum    =  $ad8a ;ausdruck,
106:    b79e             -          .eq byte      =  $b79e ;byte holen
107:    00fa             -          .eq x1l       =  $fa   ;xl, xh
108:    00fb             -          .eq x1h       =  $fb   ;und y
109:    00fc             -          .eq y1        =  $fc   ;links oben
110:    008b             -          .eq x2l       =  $8b   ;xl, xh, y
111:    008c             -          .eq x2h       =  $8c   ;rechts
112:    008d             -          .eq y2        =  $8d   ;unten
113:    009e             -          .eq adrlol    =  $9e   ;adresse
114:    009f             -          .eq adrloh    =  $9f   ;links oben
115:    008e             -          .eq adrrol    =  $8e   ;adresse
116:    008f             -          .eq adrroh    =  $8f   ;re. oben
117:    00fd             -          .eq adrlul    =  $fd   ;adresse
118:    00fe             -          .eq adrluh    =  $fe   ;links unten
119:    00b5             -          .eq adrrul    =  $b5   ;adresse
120:    00b6             -          .eq adrruh    =  $b6   ;re. unten
121:    00b7             -          .eq bitlinks  =  $b7   ;bitmuster
122:    00b8             -          .eq bitrechts=  $b8   ;li. und re.
124:    6018             -          .eq groff     =  $6018 ;grafik aus
125:    b1aa             -          .eq intfac    =  $b1aa ;int(fac)
126:    60c8             -          .eq lowtab    =  $60c8 ;zeilenanf.
127:    6190             -          .eq hightab   =  $6190 ;low u. high
128:    00b4             -          .eq rechen    =  $b4   ;rechenreg.
129:    00f9             -          .eq modus     =  $f9   ;zeichenmod.
130:    6258             -          .eq bittab    =  $6258
131:                     -;
132:                     -;start
133:                     -;=====
134:                     -;
135:    685d 20 fd ae-              jsr  komma
140:    6860 20 7e 68-              jsr  koor     ;koordinaten holen
145:    6863 78          -          sei
```

```
150:   6864 a9 34   -         lda  #52      ;prozessorport auf
155:   6866 85 01   -         sta  $01      ;ram schalten
160:   6868 20 b8 68-         jsr  horiline;horizontale linien
165:   686b e6 fc   -         inc  y1
170:   686d c6 8d   -         dec  y2
175:   686f a5 8d   -         lda  y2
180:   6871 c5 fc   -         cmp  y1
185:   6873 90 03   -         bcc  ende
189:   6875 20 9d 69-         jsr  vertiline;vertikale linien
190:   6878 a9 37   -ende     lda  #55      ;prozessorport auf
195:   687a 85 01   -         sta  $01      ;rom schalten
200:   687c 58      -         cli
205:   687d 60      -         rts           ;fertig

0:                 -;
215:                -;koordinaten holen
220:                -;=================
225:                -;
230:   687e 20 58 63-koor     jsr  $6358    ;x1,y1,modus holen
235:   6881 20 fd ae-         jsr  komma
240:   6884 20 8a ad-         jsr  frmnum   ;x-ausdehnung
245:   6887 20 aa b1-         jsr  intfac   ;nach integer
250:   688a aa      -         tax
255:   688b 98      -         tya           ;plus x1 = x2
260:   688c 18      -         clc
265:   688d 65 fa   -         adc  x1l      ;=320, dann
270:   688f 85 8b   -         sta  x2l
271:   6891 a8      -         tay
275:   6892 8a      -         txa           ;fehler
280:   6893 65 fb   -         adc  x1h
285:   6895 f0 08   -         beq  ok1
286:   6897 c9 02   -         cmp  #02
287:   6899 b0 17   -         bcs  error
295:   689b c0 40   -         cpy  #64
300:   689d b0 13   -         bcs  error
305:   689f 85 8c   -ok1      sta  x2h
310:   68a1 20 fd ae-         jsr  komma
315:   68a4 20 9e b7-         jsr  byte     ;y-ausdehnung
320:   68a7 8a      -         txa
325:   68a8 18      -         clc           ;plus y1 = y2
330:   68a9 65 fc   -         adc  y1
335:   68ab c9 c8   -         cmp  #200     ;=200, dann fehler
340:   68ad b0 03   -         bcs  error
345:   68af 85 8d   -         sta  y2
350:   68b1 60      -         rts
355:                -;
```

```
360:                  -;illegal quantity
361:                  -;================
362:                  -;
365:  68b2 20 18 60-error    jsr  groff       ;auf text schalten
370:  68b5 4c 48 b2-         jmp  $b248
371:                  -;
380:                  -;horizontale linie zeichnen
381:                  -;==========================
382:                  -;
383:  68b8 a6 fc      -horiline ldx  y1
390:  68ba a5 fa      -         lda  x1l         ;(int(x1/8))*8
395:  68bc 29 f8      -         and  #$f8
400:  68be a8         -         tay              ;a) plus zeilen-
405:  68bf 18         -         clc
410:  68c0 7d c8 60-            adc  lowtab,x    ;anfang y1
415:  68c3 85 9e      -         sta  adrlol
420:  68c5 a5 fb      -         lda  x1h         ;= adresse linke,
425:  68c7 7d 90 61-            adc  hightab,x
430:  68ca 85 9f      -         sta  adrloh      ;obere ecke
435:  68cc a6 8d      -         ldx  y2
440:  68ce 98         -         tya              ;b) plus zeilen-
445:  68cf 18         -         clc
450:  68d0 7d c8 60-            adc  lowtab,x    ;anfang y2
455:  68d3 85 fd      -         sta  adrlul
460:  68d5 a5 fb      -         lda  x1h         ;= adresse linke,
465:  68d7 7d 90 61-            adc  hightab,x
470:  68da 85 fe      -         sta  adrluh      ;untere ecke
475:  68dc a5 fa      -         lda  x1l
480:  68de 29 07      -         and  #07         ;bitmuster fuer
485:  68e0 aa         -         tax
490:  68e1 bd 3d 6a-            lda  linkstab,x ;linken rand
495:  68e4 85 b7      -         sta  bitlinks
500:  68e6 a6 fc      -         ldx  y1
505:  68e8 a5 8b      -         lda  x2l         ;(int(x2/8))*8
510:  68ea 29 f8      -         and  #$f8
515:  68ec a8         -         tay              ;a) plus zeilen-
520:  68ed 18         -         clc
525:  68ee 7d c8 60-            adc  lowtab,x    ;anfang y1
530:  68f1 85 8e      -         sta  adrrol
535:  68f3 a5 8c      -         lda  x2h         ;=adresse rechte,
540:  68f5 7d 90 61-            adc  hightab,x
545:  68f8 85 8f      -         sta  adrroh      ;obere ecke
550:  68fa a6 8d      -         ldx  y2
555:  68fc 98         -          tya             ;b) plus zeilen-
560:  68fd 18         -         clc
565:  68fe 7d c8 60-            adc  lowtab,x    ;anfang y2
```

```
570:   6901 85 b5    -          sta  adrrul
575:   6903 a5 8c    -          lda  x2h      ;=adresse rechte,
580:   6905 7d 90 61-           adc  hightab,x
585:   6908 85 b6    -          sta  adrruh   ;untere ecke
590:   690a a5 8b    -          lda  x21
595:   690c 29 07    -          and  #07      ;bitmuster fuer
600:   690e aa       -          tax
605:   690f bd 45 6a-           lda  rechtstab,x ;rechten rand
610:   6912 85 b8    -          sta  bitrechts
615:                 -;
620:   6914 a0 00    -          ldy  #00      ;adresse rechter,
625:   6916 a5 9e    -          lda  adrlol   ;rand und linker
630:   6918 c5 8e    -          cmp  adrrol   ;rand gleich,
635:   691a d0 0d    -          bne  weiter   ;dann linie nur
640:   691c a5 9f    -          lda  adrloh   ;ein byte breit
645:   691e c5 8f    -          cmp  adrroh
650:   6920 d0 07    -          bne  weiter
655:                 -;
660:                 -;linie nur ein byte breit
665:                 -;=======================
670:                 -;
675:   6922 a5 b7    -          lda  bitlinks;bitmuster ueber-
680:   6924 25 b8    -          and  bitrechts;lagern
685:   6926 4c 67 69-           jmp  plot     ;zeichnen
690:                 -;
695:                 -;linie mehr als ein byte breit
700:                 -;============================
705:                 -;
710:   6929 a5 8e    -weiter    lda  adrrol   ;adresse
715:   692b 38       -          sec
720:   692c e5 9e    -          sbc  adrlol   ;rechter rand
725:   692e 85 b4    -          sta  rechen
730:   6930 a5 8f    -          lda  adrroh   ;minus adresse
735:   6932 e5 9f    -          sbc  adrloh
740:   6934 4a       -          lsr  a        ;linker rand
745:   6935 66 b4    -          ror  rechen
750:   6937 4a       -          lsr  a        ;geteilt durch 8
755:   6938 66 b4    -          ror  rechen
760:   693a 4a       -          lsr  a        ;=anzahl zwischen-
765:   693b 66 b4    -          ror  rechen
770:   693d a6 b4    -          ldx  rechen   ;bytes
775:   693f a5 b7    -          lda  bitlinks;linken rand
780:   6941 20 67 69-           jsr  plot     ;zeichen
781:   6944 4c 4c 69-           jmp  wei4     ;zeichen
785:   6947 a9 ff    -loop      lda  #$ff     ;zwischenbyte
790:   6949 20 90 69-           jsr  plot1    ;zeichnen
```

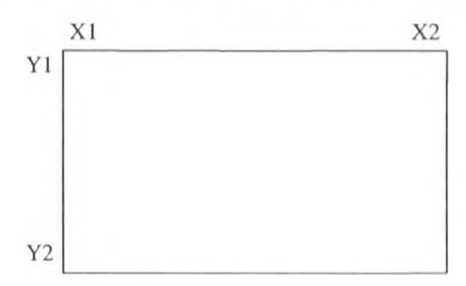
```
795:   694c a5 9e   -wei4     lda  adrlol
800:   694e 18      -         clc             ;ein byte nach
805:   694f 69 08   -         adc  #08
810:   6951 85 9e   -         sta  adrlol     ;rechts, damit
815:   6953 90 02   -         bcc  wei2
820:   6955 e6 9f   -         inc  adrloh     ;adresse um acht
825:   6957 a5 fd   -wei2     lda  adrlul
830:   6959 18      -         clc             ;zu erhoehen
835:   695a 69 08   -         adc  #08
840:   695c 85 fd   -         sta  adrlul
845:   695e 90 02   -         bcc  wei3
850:   6960 e6 fe   -         inc  adrluh     ;naechstes
855:   6962 ca      -wei3     dex
860:   6963 d0 e2   -         bne  loop       ;zwischenbyte
865:   6965 a5 b8   -         lda  bitrechts ;rechten rand
870:                -                         ;zeichnen
871:                -;
875:                -;randbyte zeichnen
880:                -;=================
881:                -;
885:   6967 24 f9   -plot     bit  modus      ;modus pruefen
890:   6969 70 0f   -         bvs  setz       ;zeichnen
895:   696b 30 18   -         bmi  inv        ;invertieren
900:   696d 49 ff   -loesch   eor  #$ff       ;loeschen
901:   696f 48      -         pha
905:   6970 31 fd   -         and  (adrlul),y ;untere linie
910:   6972 91 fd   -         sta  (adrlul),y
915:   6974 68      -         pla
920:   6975 31 9e   -         and  (adrlol),y ;obere linie
925:   6977 91 9e   -         sta  (adrlol),y
930:   6979 60      -         rts
931:                -;
940:   697a 48      -setz     pha
945:   697b 11 fd   -         ora  (adrlul),y ;untere linie
950:   697d 91 fd   -         sta  (adrlul),y
955:   697f 68      -         pla
960:   6980 11 9e   -         ora  (adrlol),y
965:   6982 91 9e   -         sta  (adrlol),y ;obere linie
970:   6984 60      -         rts
971:                -;
975:   6985 48      -inv      pha
980:   6986 51 9e   -         eor  (adrlol),y ;obere linie
985:   6988 91 9e   -         sta  (adrlol),y
990:   698a 68      -         pla
995:   698b 51 fd   -         eor  (adrlul),y ;untere linie
1000:  698d 91 fd   -         sta  (adrlul),y
```

```
1001:  698f 60      -           rts
1002:               -;
1003:               -;zwischenbyte zeichnen
1004:               -;====================
1005:               -;
1009:  6990 24 f9   -plot1      bit  modus     ;modus pruefen
1010:  6992 70 04   -           bvs  setz1     ;setzen
1015:  6994 30 ef   -           bmi  inv       ;invertieren
1020:               -;
1021:  6996 a9 00   -loesch1    lda  #00
1022:  6998 91 9e   -setz1      sta  (adrlol),y
1023:  699a 91 fd   -           sta  (adrlul),y
1024:  699c 60      -           rts
1025:               -;
1030:               -;vertikale linie zeichnen
1035:               -;========================
1040:               -;
1045:  699d a5 fc   -vertilinelda  y1
1046:  699f 29 f8   -           and  #$f8
1047:  69a1 aa      -           tax
1050:  69a2 a5 fa   -           lda  x11
1051:  69a4 a8      -           tay
1055:  69a5 29 f8   -           and  #$f8      ;(int(x1/8))*8
1060:  69a7 18      -           clc
1065:  69a8 7d c8 60-           adc  lowtab,x;plus zeilen-
1070:  69ab 85 9e   -           sta  adrlol
1075:  69ad a5 fb   -           lda  x1h       ;anfang y1 =
1080:  69af 7d 90 61-           adc  hightab,x
1085:  69b2 85 9f   -           sta  adrloh    ;adresse links
1095:  69b4 b9 58 62-           lda  bittab,y;bitmuster links
1100:  69b7 85 b7   -           sta  bitlinks;merken
1110:  69b9 a5 8b   -           lda  x21
1111:  69bb a8      -           tay
1115:  69bc 29 f8   -           and  #$f8      ;(int(x2/8))*8
1120:  69be 18      -           clc
1125:  69bf 7d c8 60-           adc  lowtab,x;plus zeilen-
1130:  69c2 85 8e   -           sta  adrrol
1135:  69c4 a5 8c   -           lda  x2h       ;anfang y1 =
1140:  69c6 7d 90 61-           adc  hightab,x
1145:  69c9 85 8f   -           sta  adrroh    ;adresse links
1155:  69cb b9 58 62-           lda  bittab,y;bitmuster rechts
1160:  69ce 85 b8   -           sta  bitrechts ;merken
1165:  69d0 a5 fc   -           lda  y1        ;position im
1170:  69d2 29 07   -           and  #07       ;block
1175:  69d4 a8      -           tay
1175:  69d5 8c 3c 03-           sty  $033c     ;als zaehler
```

```
69d8 a5 8d    -         lda  y2
69da 38       -         sec              ;(y2-y1)+1 =
69db e5 fc    -         sbc  y1
69dd aa       -         tax              ;anzahl zwischen-
69de e8       -         inx
69df 8e 3d 03-          stx  $033d       ;bytes
69e2 20 0c 6a-loop1     jsr  plot2       ;linken rand
69e5 ca       -         dex
69e6 f0 23    -         beq  out         ;fertig
69e8 c8       -         iny              ;grafikblock
69e9 c0 08    -         cpy  #08         ;beendet
69eb 90 f5    -         bcc  loop1       ;nein, weiter
69ed a5 9e    -         lda  adrlol
69ef 18       -         clc              ;ja, naechster
69f0 69 40    -         adc  #$40
69f2 85 9e    -         sta  adrlol      ;block, d.h.
69f4 a5 9f    -         lda  adrloh
69f6 69 01    -         adc  #01         ;adresse muss
69f8 85 9f    -         sta  adrloh
69fa a5 8e    -         lda  adrrol      ;um 320 erhoeht
69fc 18       -         clc
69fd 69 40    -         adc  #$40        ;werden
69ff 85 8e    -         sta  adrrol
6a01 a5 8f    -         lda  adrroh
6a03 69 01    -         adc  #01
6a05 85 8f    -         sta  adrroh
6a07 a0 00    -         ldy  #00
6a09 f0 d7    -         beq  loop1       ;immer
6a0b 60       -out      rts              ;ende
              -;
              -;zeichnet linie
              -;==============
              -;
6a0c 24 f9    -plot2    bit  modus       ;modus pruefen
6a0e 70 13    -         bvs  setz2       ;setzen
6a10 30 1e    -         bmi  inv2        ;invertieren
6a12 a5 b7    -loesch2  lda  bitlinks
6a14 49 ff    -         eor  #$ff
6a16 31 9e    -         and  (adrlol),y ;linke linie
6a18 91 9e    -         sta  (adrlol),y
6a1a a5 b8    -         lda  bitrechts
6a1c 49 ff    -         eor  #$ff
6a1e 31 8e    -         and  (adrrol),y ;rechte linie
6a20 91 8e    -         sta  (adrrol),y
6a22 60       -         rts
              -;
```

```
1395:  6a23 a5 b7   -setz2      lda  bitlinks
1400:  6a25 11 9e   -           ora  (adrlol),y ;linke linie
1405:  6a27 91 9e   -           sta  (adrlol),y
1410:  6a29 a5 b8   -           lda  bitrechts
1415:  6a2b 11 8e   -           ora  (adrrol),y ;rechte linie
1420:  6a2d 91 8e   -           sta  (adrrol),y
1421:  6a2f 60      -           rts
1422:               -;
1425:  6a30 a5 b7   -inv2       lda  bitlinks
1430:  6a32 51 9e   -           eor  (adrlol),y ;linke linie
1435:  6a34 91 9e   -           sta  (adrlol),y
1440:  6a36 a5 b8   -           lda  bitrechts
1445:  6a38 51 8e   -           eor  (adrrol),y ;rechte linie
1450:  6a3a 91 8e   -           sta  (adrrol),y
1451:  6a3c 60      -           rts
1452:               -;
3000:  6a3d ff 7f 3f-linkstab .by$ff,$7f,$3f,$1f
3001:  6a41 0f 07 03-         .by$0f,$07,$03,$01
3002:  6a45 80 c0 e0-rechtstab.by$80,$c0,$e0,$f0
3003:  6a49 f8 fc fe-         .by$f8,$fc,$fe,$ff
```

Eine Sache, die sehr wichtig ist und bislang praktisch von keiner professionellen Erweiterung bedacht wurde, möchte ich hier noch ansprechen. Dies ist die Frage, ob die vier Ecken doppelt jeweils sowohl von der waagerechten als auch von der senkrechten Linie gezeichnet werden dürfen. Wenn man die Koordinaten der Ecken z.B. mit X1,Y1;X1,Y2;X2,Y1;X2,Y2 bezeichnet, scheint es egal zu sein, ob man die senkrechte Linie von Y1 nach Y2 oder von Y1+1 nach Y2–1 zeichnet, wenn der Eckpunkt bereits von der horizontalen Linie abgedeckt wurde:



Leider gilt das Motto »doppelt hält besser« nur so lange, bis man ein Rechteck invertieren möchte. Dann nämlich werden die Eckpunkte doppelt invertiert, was heißt, daß sie denselben Zustand annehmen wie vor Ausführung des Befehls. Man darf daher die Eckpunkte tatsächlich nur einmal bearbeiten, alles andere ist ein grober Programmierfehler. Daher zeichnet die obige Routine die vertikale Linie immer nur von Y1+1 nach Y2–1, die horizontale jedoch von X1 nach X2. Man könnte die Sache auch umdrehen, also die waagerechte Linie von X1+1 nach X2–1 zeichnen, dafür jedoch die senkrechte von Y1 nach Y2.

Aufgerufen wird das Programm mit `SYS 26717,X,Y,M,DX,DY.`

Dabei bedeuten:
X: X-Koordinate der linken, oberen Ecke
Y: Y-Koordinate der linken, oberen Ecke
M: Zeichenmodus (32, 64 oder 128 s.o.)
DX: Ausbreitung des Rechtecks in X-Richtung
DY: Ausbreitung des Rechtecks in Y-Richtung

4.4 Zeichnen von Kreisen/Ellipsen

Auch beim Erstellen von Ellipsen oder Kreisen tun sich viele Grafikerweiterungen sehr schwer. Um die Gründe hierfür herauszufinden, muß man sich erst einmal den Aufbau einer Ellipse ansehen:

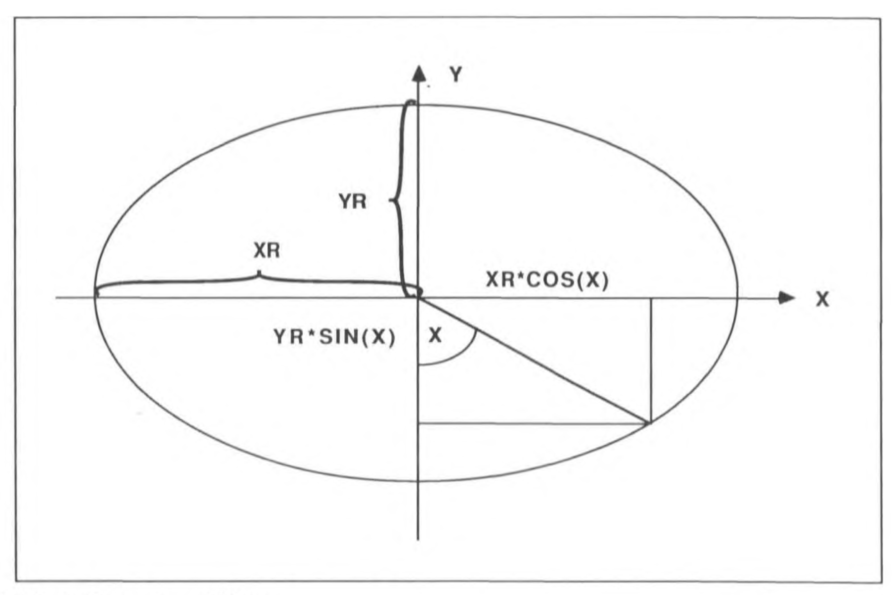


Bild 4.3: *Aufbau einer Ellipse*

Um die Koordinaten der einzelnen Punkte zu berechnen, gilt folgendes: Wenn man den Mittelpunkt der Ellipse durch die Koordinaten XM und YM beschreibt, den Radius in X-Richtung mit XR und denselben in Y-Richtung mit YR bezeichnet, und schließlich den Winkel W einführt, kann man eine Ellipse als die Punktmenge definieren, die diese Gleichungen erfüllt:

X = XM + XR*COS(W) 0 Grad ⇐W ⇐360 Grad
Y = YM + YR*SIN(W) 0 Grad ⇐W ⇐360 Grad

Falls XR und YR gleich sind, liegt der Sonderfall eines Kreises vor. Man braucht also tatsächlich nur eine Routine, um sowohl Kreise als auch Ellipsen zeichnen zu können.

Die meisten Grafikerweiterungen gehen genau so vor, wie es die Formeln vorschreiben: Sie berechnen für jeden Winkel die entsprechenden Sinus- und Cosinus-Werte im FAC, multiplizieren die Radien hinein und wandeln das Ergebnis in eine Integer-Zahl um, die sie zu den Mittelpunktkoordinaten hinzuaddieren. Ein Problem besteht darin, daß bei einer idealen Ellipse zwei nebeneinanderliegende Winkel W unendlich dicht zusammen liegen, d.h. es sind unendlich viele Berechnungsschritte erforderlich. Die Grafikerweiterungen helfen sich im allgemeinen damit, daß sie natürlich nur einige Punkte berechnen und diese dann durch Linien verbinden. Je größer diese sogenannte Schrittweite ist, desto schneller kann die Ellipse aufgebaut werden, desto größer ist aber auch die Abweichung von einer idealen Ellipse.

Es ist klar, daß man auf dem C64 keine solche ideale Ellipse darstellen kann, da die Grafikauflösung nicht unendlich groß ist, sondern nur 320 * 200 Punkte beträgt. Die genaueste mögliche Ellipse erhält man, wenn zwei nebeneinanderliegende X- **und** Y-Werte jeweils nicht mehr als einen Grafikpunkt auseinanderliegen. Dies ist das C64-Kriterium, an dem man den Begriff »Genauigkeit« definitiv festhalten kann.

In der Praxis heißt dies, daß man eine Schrittweite von ca. 0.1 Grad nehmen muß, um dieser Forderung zu entsprechen. Leider führt dies aber dazu, daß es bei manchen Erweiterungen mehr als zehn Sekunden dauern kann, bevor eine Ellipse gezeichnet wurde! Da dies nicht akzeptabel erscheint, überlegte man sich, wie man den Zeichenvorgang optimieren könnte.

Zunächst kann man die mathematische Beziehung zwischen den Sinus- und Cosinus-Werten bestimmter Winkel ausnutzen. Hierbei gilt:

1:SIN(W) = SIN(180 Grad-W) = –SIN(180 Grad+W)= – SIN(360 Grad-W)
2:COS(W) = COS(360 Grad-W)= –COS(180 Grad-W)= – COS(180 Grad+W)

Durch diese Tatsache ist es möglich, mit einem berechneten Wertepaar jeweils in jedem Quadranten einen Punkt zu setzen, da sie sich nur im Vorzeichen unterscheiden. Wenn wir den Term XR*COS(W) mit XOFF und YR*SIN(W) mit YOFF bezeichnen, ergeben sich für die berechneten Koordinaten der einzelnen Quadranten folgende Zusammenhänge:

1. Quadrant (0 – 90 Grad): X = XM + XOFF ; Y = YM – YOFF
2. Quadrant (90 – 180 Grad): X = XM + XOFF ; Y = YM + YOFF
3. Quadrant (180 – 270 Grad): X = XM – XOFF ; Y = YM + YOFF
4. Quadrant (270 – 360 Grad): X = XM – XOFF ; Y = YM – YOFF

Obwohl diese Methode die Rechenzeit um den Faktor 4 verringern kann, weist sie auch einen Nachteil auf, mit dem wir noch schwer zu kämpfen haben werden: Während es nach der ursprünglichen Methode möglich war, beliebige Ellipsenausschnitte zu zeichnen, können wir jetzt nur noch Vollellipsen darstellen. Durch relativ komplizierte Programmierung werden wir jedoch später diesen Nachteil kompensieren.

Trotz dieses Kunstgriffs ist es aber nicht gelungen, die Rechenzeit so weit zu reduzieren, daß sie uns brauchbar erschiene. Das eigentliche Problem besteht in der Verwendung der Fließkommaoperationen. Eigentlich ist deren Verwendung als paradox zu bezeichnen, da bei der Grafik ja nur Integer-Werte benutzt werden können. Wie soll es aber sonst möglich sein, die Cosinus- und Sinus-Werte zu berechnen?

Die entscheidende Antwort ist: überhaupt nicht. Die Idee unserer Routine besteht vielmehr darin, diese Werte in einer Tabelle anzulegen, was den entscheidenden Schritt bedeutet. Da die Sinus- bzw. Cosinus-Werte nur im Bereich von 0 bis 1 liegen, müssen wir sie mit einem Faktor multiplizieren, der eine Umwandlung ins Integer-Format erlaubt. Da wir natürlich später wieder durch diesen Faktor teilen müssen, sollte er möglichst »prozessorfreundlich« sein: Wir werden den Wert 128 benutzen. Um durch ihn zu teilen, braucht man nur die Bits in einem Byte siebenmal nach rechts zu verschieben. Für unsere Tabelle gilt daher:

```
Wert = INT(COS(W)*128)
```

Wenn wir dann den Offset berechnen wollen, müssen wir

```
XOFF = XM + XR*(Wert/128)
```

rechnen. An dieser Stelle müssen wir uns leider wieder mit dem Problem der Genauigkeit auseinandersetzen. Wieviele Werte soll unsere Tabelle enthalten und in welcher Schrittweite sollen diese angelegt sein?

Wir benötigen die Werte von 0 Grad <=W <90 Grad, mit denen wir ja auch die anderen drei Quadranten erschlagen können (s.o.). Unsere Forderung ist die, daß zwei benachbarte Werte um maximal die Zahl 1 differieren dürfen, um die größtmögliche darstellbare Genauigkeit auf dem C64 zu erreichen. Hier hilft nur ausprobieren: Es zeigt sich, daß die Schrittweite 1/195 die größte mögliche ist, die diese Forderung erfüllt. Eine geringere Schrittweite bringt einen Zuwachs an Rechenzeit, eine größere einen Verlust an Genauigkeit. Wir haben nach dem Grundsatz »so grob wie möglich, so fein wie nötig« die optimale Schrittweite erhalten. Unsere Tabellenwerte können wir nun in einer Schleife berechnen:

```
FOR I = 0 TO 195: Wert(I) = 128*INT(COS((90/195)*I)): NEXT I
```

Wir haben es also geschafft, den Übergang vom Winkel W in eine Schleife von Werten zu erreichen. Dies ist ganz entscheidend für die Ausführungszeit der Routine, da nun keinerlei Winkelberechnungen mehr durchgeführt werden müssen. Die kritischste Stelle der Genauigkeit liegt übrigens beim Cosinus bei 90 Grad, da er dort seine größte Steigung aufweist, beim Sinus bei 0 Grad.

Wie können wir nun auf einfachste Weise den X-Offset berechnen? Wir müssen einfach in einer Schleife von 0 bis 195 die jeweiligen Werte auslesen, mit dem X-Radius multiplizieren und durch 128 teilen:

```
FOR I = 0 TO 195: X = XM + XR*(Wert(I)/128): NEXT I
```

Hier der Beweis: Wenn wir die Berechnungsgrundlage für die Tabellenwerte einsetzen, ergibt sich folgendes:

```
X = XM  + XR*(128*INT(COS(90/195)*I)/128)
```

Dies kann man aber auch so ausdrücken:

```
FOR I = 0 TO 90: X = XM + XR*INT(COS(I)): NEXT I
```

Das einzige Problem besteht darin, daß nur Radien bis zum Wert 128 einschließlich zugelassen werden können. Bis zu diesem Zeitpunkt nämlich kann der Ausdruck

```
XOFF = XR*(Wert(I)/128)
```

nicht größer als eins werden, d.h. unsere gewünschte Genauigkeit ist gegeben. Bei größeren Radien wäre dies nicht mehr der Fall. Mit dieser Einschränkung kann man aber leben, wenn man bedenkt, daß in Y-Richtung nur 200 Punkte zur Verfügung stehen, aber 2*128=256 Punkte zulässig sind.

Für die Sinus-Werte brauchen wir nicht extra eine Tabelle anzulegen, da gilt:

```
SIN(W) = COS(90-W)
```

Wir können also unsere Tabelle einmal von vorne für die Cosinus-Werte benutzen und einmal von hinten für den Sinus auslesen. Nun stellt die Umsetzung in ein Programm kein Problem mehr dar. In einer Schleife von 0 bis 195 wird jeweils für alle vier Quadranten ein Punkt berechnet. Hier erkennt man einen weiteren Vorteil der Routine: Die Ausführungszeit ist völlig unabhängig von Größe und Form der Ellipse.

Nun zu dem eingangs geschilderten Problem, dem Erstellen eines Ellipsenausschnitts. Hierfür wurden zusätzliche Routinen installiert, die genauso funktionieren wie die Routine für die ganzen Ellipsen, aber jeweils nur einen Punkt in einem Quadranten zeichnen. Je nach Lage von Anfangs- und Endpunkt des Ausschnitts werden diese Routinen nun nacheinander kombiniert aufgerufen. Nun aber zu dem Quelltext dieser superschnellen Routine:

Listing: »kreis/ellipse«

```
638b     -;kreis/ellipse
638b     -          .ba $638b
         -;
aefd     -          .eq komma    = $aefd ;komma
ad8a     -          .eq frmnum   = $ad8a ;ausdruck,
b79e     -          .eq byte     = $b79e ;byte holen
00fa     -          .eq xl       = $fa   ;x-koor.low
```

```
00fb          -          .eq xh       =  $fb   ;x-koor.high
00fc          -          .eq y        =  $fc   ;y-koordin.
00fd          -          .eq xr       =  $fd   ;x-radius
008b          -          .eq yr       =  $8b   ;y-radius
008c          -          .eq xoff     =  $8c   ;x-offset
008d          -          .eq yoff     =  $8d   ;y-offset
00fe          -          .eq zaehl    =  $fe   ;zaehler
008e          -          .eq x1       =  $8e   ;mittelpunkt
008f          -          .eq x2       =  $8f   ;x-koord.
0071          -          .eq mu       =  $71   ;rechenreg.
00b4          -          .eq anqua    =  $b4   ;anf.-u.end-
00b5          -          .eq endqua   =  $b5   ;quadranten
00b6          -          .eq anpunkt  =  $b6   ;startpunkt
0072          -          .eq endpunkt =  $72   ;endpunkt
609c          -          .eq plot     =  $609c ;setz punkt
6018          -          .eq groff    =  $6018 ;grafik aus
bc0c          -          .eq facarg   =  $bc0c ;arg=fac
ba28          -          .eq memmult  =  $ba28 ;fac=k*fac
b1aa          -          .eq intfac   =  $b1aa ;int(fac)
bba2          -          .eq memfac   =  $bba2 ;fac=konst.
b850          -          .eq memmin   =  $b850 ;fac=k-fac
bbd4          -          .eq facmem   =  $bbd4 ;konst.=fac
                         -;
                         -;start
                         -;=====
                         -;
638b 20 fd ae-           jsr  komma
638e 20 ea 66-           jsr  koor       ;koordinaten holen
6391 a5 b4   -           lda  anqua      ;start im ersten
6393 c9 01   -           cmp  #01        ;quadranten
6395 d0 14   -           bne  ausschnitt;mit erstem punkt
6397 a5 b6   -           lda  anpunkt
6399 c9 01   -           cmp  #01        ;nein, nur kreis-
639b d0 0e   -           bne  ausschnitt;abschnitt
639d a5 b5   -           lda  endqua     ;ende im letzten
639f c9 04   -           cmp  #04        ;quadranten mit
63a1 d0 08   -           bne  ausschnitt;letztem punkt
63a3 a5 72   -           lda  endpunkt
63a5 c9 c3   -           cmp  #195       ;nein, nur kreis-
63a7 d0 02   -           bne  ausschnitt;abschnitt
63a9 f0 03   -           beq  ganz       ;ja, ganzer kreis !
                         -;
63ab 4c 5a 64-ausschnitjmp  aussch
                         -;
                         -;ganzen kreis/ellipse zeichen
                         -;===========================
```

```
                   -;
 63ae a5 fa        -ganz       lda  xl
 63b0 85 8e        -           sta  x1           ;mittelpunkt
 63b2 a5 fb        -           lda  xh           ;koordinaten in x
 63b4 85 8f        -           sta  x2           ;merken
 63b6 a9 01        -           lda  #01
 63b8 85 ff        -           sta  zaehl+1
 63ba a9 c3        -           lda  #195
 63bc 85 fe        -           sta  zaehl        ;zaehler setzen
 63be 78           -           sei
 63bf a9 34        -           lda  #52          ;prozessorport
 63c1 85 01        -           sta  $01          ;auf ram schalten
                   -;
                   -;berechnung von je 4 punkten
                   -;===========================
                   -;
 63c3 a6 fe        -loop       ldx  zaehl        ;zaehler holen
 63c5 bd 98 67-                lda  cos-1,x      ;cosinus-wert holen
 63c8 a6 fd        -           ldx  xr
 63ca 20 85 67-                jsr  mult         ;*xr/128 berechnen
 63cd 85 8c        -           sta  xoff         ;=x-offset
 63cf a6 ff        -           ldx  zaehl+1      ;zaehler holen
 63d1 bd 98 67-                lda  cos-1,x      ;sinus wert holen
 63d4 a6 8b        -           ldx  yr
 63d6 20 85 67-                jsr  mult         ;*yr/128 brechnen
 63d9 85 8d        -           sta  yoff         ;=y-offset
 63db a5 8e        -           lda  x1           ;mittelpunkts-
 63dd a4 8f        -           ldy  x2           ;koordinaten holen
 63df 18           -           clc               ;1. punkt
 63e0 65 8c        -           adc  xoff         ;x=x+xoff
 63e2 90 01        -           bcc  l1
 63e4 c8           -           iny
 63e5 c0 00        -l1         cpy  #00          ;xh=0, dann o.k.
 63e7 f0 08        -           beq  l3
 63e9 c0 02        -           cpy  #02          ;xh1, dann kein
 63eb b0 26        -           bcs  l4           ;punkt setzen
 63ed c9 40        -           cmp  #64          ;xh=1 und xl=64,
 63ef b0 22        -           bcs  l4           ;dann kein punkt
 63f1 85 fa        -l3         sta  xl
 63f3 84 fb        -           sty  xh
 63f5 a5 fc        -           lda  y            ;y=y+yoff
 63f7 18           -           clc
 63f8 65 8d        -           adc  yoff
 63fa b0 0a        -           bcs  l2
 63fc c9 c8        -           cmp  #200         ;y=200, dann kein
 63fe b0 06        -           bcs  l2           ;punkt setzen
```

```
6400 aa       -      tax             ;y-koordinate
6401 a4 fa    -      ldy   xl        ;x-koordinate low
6403 20 9c 60-       jsr   plot      ;punkt setzen
6406 a5 fc    -12    lda   y
6408 38       -      sec             ;2.punkt
6409 e5 8d    -      sbc   yoff
640b 90 06    -      bcc   14        ;y=y-yoff
640d aa       -      tax             ;y-koordinate
640e a4 fa    -      ldy   xl        ;x-koordinate low
6410 20 9c 60-       jsr   plot      ;punkt setzen
6413 a5 8e    -14    lda   x1        ;mittelpunkts-
6415 a4 8f    -      ldy   x2        ;koordinaten holen
6417 38       -      sec
6418 e5 8c    -      sbc   xoff      ;3. punkt
641a b0 01    -      bcs   15        ;x=x-xoff
641c 88       -      dey
641d c0 00    -15    cpy   #00       ;xh=0, dann o.k.
641f f0 08    -      beq   16
6421 c0 02    -      cpy   #02       ;xh1, dann kein
6423 b0 26    -      bcs   17        ;punkt  setzen
6425 c9 40    -      cmp   #64       ;xh=1 und xl=64,
6427 b0 22    -      bcs   17        ;dann kein punkt
6429 85 fa    -16    sta   xl
642b 84 fb    -      sty   xh
642d a5 fc    -      lda   y         ;y=y+yoff
642f 18       -      clc
6430 65 8d    -      adc   yoff
6432 b0 0a    -      bcs   18
6434 c9 c8    -      cmp   #200      ;y=200, dann kein
6436 b0 06    -      bcs   18        ;punkt setzen
6438 aa       -      tax             ;y-koordinate
6439 a4 fa    -      ldy   xl        ;x-koordinate low
643b 20 9c 60-       jsr   plot      ;punkt setzen
643e a5 fc    -18    lda   y
6440 38       -      sec             ;4.punkt
6441 e5 8d    -      sbc   yoff
6443 90 06    -      bcc   17        ;, kein punkt
6445 aa       -      tax             ;y-koordinate
6446 a4 fa    -      ldy   xl        ;x-koordinate low
6448 20 9c 60-       jsr   plot
644b e6 ff    -17    inc   zaehl+1
644d c6 fe    -      dec   zaehl
644f f0 03    -      beq   out       ;fertig
6451 4c c3 63-       jmp   loop      ;naechste 4 punkte
6454 a9 37    -out   lda   #55
6456 85 01    -      sta   $01       ;auf rom schalten
```

```
6458 58       -          cli
6459 60       -          rts
              -;
              -;kreis/ellipsenabschnitt zeichnen
              -;=================================
              -;
645a a2 c3    -aussch    ldx  #195
645c 86 fe    -          stx  zaehl
645e a2 01    -          ldx  #01        ;tabelle
6460 86 ff    -          stx  zaehl+1
6462 78       -          sei             ;prozessorport
6463 a9 34    -          lda  #52        ;auf ram
6465 85 01    -          sta  $01
6467 a6 fe    -ausloop   ldx  zaehl      ;der offsets
6469 bd 98 67-           lda  cos-1,x
646c a6 fd    -          ldx  xr         ;in x-
646e 20 85 67-           jsr  mult
6471 a6 ff    -          ldx  zaehl+1    ;und y-
6473 9d 00 d0-           sta  $d000,x
6476 bd 98 67-           lda  cos-1,x    ;richtung
6479 a6 8b    -          ldx  yr
647b 20 85 67-           jsr  mult       ;aufbauen
647e a6 ff    -          ldx  zaehl+1
6480 9d 00 d1-           sta  $d100,x
6483 e6 ff    -          inc  zaehl+1
6485 c6 fe    -          dec  zaehl
6487 d0 de    -          bne  ausloop
              -;
6489 a5 72    -          lda  endpunkt   ;wenn endpunkt=1,
648b c9 01    -          cmp  #01        ;=2 setzen, da
648d d0 02    -          bne  weiter     ;sonst fehler
648f a9 02    -          lda  #02        ;auftreten
6491 85 72    -weiter    sta  endpunkt
6493 a5 fa    -          lda  xl
6495 85 8e    -          sta  x1
6497 a5 fb    -          lda  xh
6499 85 8f    -          sta  x2
649b a5 b4    -          lda  anqua      ;anfangsquadrant
649d 0a       -          asl  a
649e 0a       -          asl  a          ;*4
649f 18       -          clc
64a0 65 b5    -          adc  endqua     ;plus endquadrant
64a2 aa       -          tax             ;=tabellenoffset
64a3 bd 9c 65-           lda  lowtab,x
64a6 8d b0 64-           sta  start+1    ;startadresse
64a9 bd b1 65-           lda  hightab,x
```

```
64ac 8d b1 64-           sta  start+2     ;holen
64af 20 00 00-start      jsr  $0000       ;routine ausfuehren
64b2 a9 37    -          lda  #55         ;prozessorport
64b4 85 01    -          sta  $01         ;auf rom schalten
64b6 58       -          cli
64b7 60       -          rts              ;ende
              -;
              -;
              -;start im 1., ende im 1.quadranten
              -;
64b8 a5 b6    -s1e1      lda  anpunkt
64ba c5 72    -          cmp  endpunkt
64bc 90 0a    -          bcc  s1e11
64be a2 c3    -          ldx  #195
64c0 20 c6 65-           jsr  erster
64c3 a9 01    -          lda  #01
64c5 4c 0e 65-           jmp  s2e1+2
64c8 a6 72    -s1e11     ldx  endpunkt
64ca 4c c6 65-           jmp  erster
              -;
              -;start im 1., ende im 2.quadranten
              -;
64cd a5 b6    -s1e2      lda  anpunkt
64cf a2 c3    -          ldx  #195
64d1 20 c6 65-           jsr  erster
64d4 a9 01    -          lda  #01
64d6 a6 72    -          ldx  endpunkt
64d8 4c 14 66-           jmp  zweiter
              -;
              -;start im 1., ende im 3.quadranten
              -;
64db a5 b6    -s1e3      lda  anpunkt
64dd a2 c3    -          ldx  #195
64df 20 c6 65-           jsr  erster
64e2 a9 01    -          lda  #01
64e4 a2 c3    -          ldx  #195
64e6 20 14 66-           jsr  zweiter
64e9 a9 01    -          lda  #01
64eb a6 72    -          ldx  endpunkt
64ed 4c 58 66-           jmp  dritter
              -;
              -;start im 1., ende im 4.quadranten
              -;
64f0 a5 b6    -s1e4      lda  anpunkt
64f2 a2 c3    -          ldx  #195
64f4 20 c6 65-           jsr  erster
```

```
64f7 a9 01   -          lda  #01
64f9 a2 c3   -          ldx  #195
64fb 20 14 66-          jsr  zweiter
64fe a9 01   -          lda  #01
6500 a2 c3   -          ldx  #195
6502 20 58 66-          jsr  dritter
6505 a9 01   -          lda  #01
6507 a6 72   -          ldx  endpunkt
6509 4c aa 66-          jmp  vierter
             -;
             -;start im 2., ende im 1.quadranten
             -;
650c a5 b6   -s2e1      lda  anpunkt
650e a2 c3   -          ldx  #195
6510 20 14 66-          jsr  zweiter
6513 a9 01   -          lda  #01
6515 a2 c3   -          ldx  #195
6517 20 58 66-          jsr  dritter
651a a9 01   -          lda  #01
651c a2 c3   -          ldx  #195
651e 20 aa 66-          jsr  vierter
6521 a9 01   -          lda  #01
6523 4c c8 64-          jmp  s1e1+16
             -;
             -;start im 2., ende im 2.quadranten
             -;
6526 a5 b6   -s2e2      lda  anpunkt
6528 c5 72   -          cmp  endpunkt
652a 90 0a   -          bcc  s2e22
652c a2 c3   -          ldx  #195
652e 20 14 66-          jsr  zweiter
6531 a9 01   -          lda  #01
6533 4c 4a 65-          jmp  s3e2+2
6536 4c d6 64-s2e22     jmp  s1e2+9
             -;
             -;start im 2., ende im 3.quadranten
             -;
6539 a5 b6   -s2e3      lda  anpunkt
653b 4c e4 64-          jmp  s1e3+9
             -;
             -;start im 2., ende im 4.quadranten
             -;
653e a5 b6   -s2e4      lda  anpunkt
6540 4c f9 64-          jmp  s1e4+9
             -;
             -;start im 3., ende im 1.quadranten
```

```
                  -;
6543 a5 b6    -s3e1      lda  anpunkt
6545 4c 15 65-           jmp  s2e1+9
                  -;
                  -;start im 3., ende im 2.quadranten
                  -;
6548 a5 b6    -s3e2      lda  anpunkt
654a a2 c3    -          ldx  #195
654c 20 58 66-           jsr  dritter
654f a9 01    -          lda  #01
6551 a2 c3    -          ldx  #195
6553 20 aa 66-           jsr  vierter
6556 a9 01    -          lda  #01
6558 4c cf 64-           jmp  s1e2+2
                  -;
                  -;start im 3., ende im 3.quadranten
                  -;
655b a5 b6    -s3e3      lda  anpunkt
655d c5 72    -          cmp  endpunkt
655f 90 0a    -          bcc  s3e33
6561 a2 c3    -          ldx  #195
6563 20 58 66-           jsr  dritter
6566 a9 01    -          lda  #01
6568 4c 7f 65-           jmp  s4e3+2
656b 4c eb 64-s3e33      jmp  s1e3+16
                  -;
                  -;start im 3., ende im 4.quadranten
                  -;
656e a5 b6    -s3e4      lda  anpunkt
6570 4c 00 65-           jmp  s1e4+16
                  -;
                  -;start im 4., ende im 1.quadranten
                  -;
6573 a5 b6    -s4e1      lda  anpunkt
6575 4c 1c 65-           jmp  s2e1+16
                  -;
                  -;start im 4., ende im 2.quadranten
                  -;
6578 a5 b6    -s4e2      lda  anpunkt
657a 4c 51 65-           jmp  s3e2+9
                  -;
                  -;start im 4., ende im 3.quadranten
                  -;
657d a5 b6    -s4e3      lda  anpunkt
657f a2 c3    -          ldx  #195
6581 20 aa 66-           jsr  vierter
```

```
6584 a9 01   -            lda   #01
6586 4c dd 64-            jmp   s1e3+2
             -;
             -;start im 4., ende im 4.quadranten
             -;
6589 a5 b6   -s4e4        lda   anpunkt
658b c5 72   -            cmp   endpunkt
658d 90 0a   -            bcc   s4e44
658f a2 c3   -            ldx   #195
6591 20 aa 66-            jsr   vierter
6594 a9 01   -            lda   #01
6596 4c f2 64-            jmp   s1e4+2
6599 4c 07 65-s4e44       jmp   s1e4+23
             -;
659c 00 00 00-lowtab      .by0,0,0,0,0
65a1 b8 cd db-            .by(s1e1),(s1e2),(s1e3),(s1e4)
65a5 0c 26 39-            .by(s2e1),(s2e2),(s2e3),(s2e4)
65a9 43 48 5b-            .by(s3e1),(s3e2),(s3e3),(s3e4)
65ad 73 78 7d-            .by(s4e1),(s4e2),(s4e3),(s4e4)
65b1 00 00 00-hightab     .by0,0,0,0,0
65b6 64 64 64-            .by(s1e1),(s1e2),(s1e3),(s1e4)
65ba 65 65 65-            .by(s2e1),(s2e2),(s2e3),(s2e4)
65be 65 65 65-            .by(s3e1),(s3e2),(s3e3),(s3e4)
65c2 65 65 65-            .by(s4e1),(s4e2),(s4e3),(s4e4)
             -;
             -;ersten quadranten zeichnen
             -;==========================
             -;
65c6 85 fe   -erster      sta   zaehl
65c8 86 ff   -            stx   zaehl+1
65ca a9 c3   -            lda   #195          ;zaehler setzen
65cc 38      -            sec
65cd e5 fe   -            sbc   zaehl         ;und invertieren
65cf 85 fe   -            sta   zaehl
65d1 a9 c3   -            lda   #195
65d3 38      -            sec
65d4 e5 ff   -            sbc   zaehl+1
65d6 85 ff   -            sta   zaehl+1
65d8 a6 ff   -loopq1      ldx   zaehl+1       ;zaehler holen
65da bd 00 d0-            lda   $d000,x       ;x-odffset holen
65dd 85 8c   -            sta   xoff          ;merken
65df bd 00 d1-            lda   $d100,x       ;y-offset holen
65e2 85 8d   -            sta   yoff          ;merken
65e4 a5 8e   -            lda   x1            ;mittelpunkts-
65e6 a4 8f   -            ldy   x2            ;koordinaten holen
65e8 18      -            clc
```

```
65e9 65 8c    -          adc  xoff        ;x=x+xoff
65eb 90 01    -          bcc  lq11
65ed c8       -          iny
65ee c0 00    -lq11      cpy  #00         ;xh=0, dann o.k.
65f0 f0 08    -          beq  lq13
65f2 c0 02    -          cpy  #02         ;xh1, dann kein
65f4 b0 15    -          bcs  lq12        ;punkt setzen
65f6 c9 40    -          cmp  #64         ;xh=1 und xl=64,
65f8 b0 11    -          bcs  lq12        ;dann kein punkt
65fa 85 fa    -lq13      sta  xl
65fc 84 fb    -          sty  xh
65fe a5 fc    -          lda  y           ;y=y-yoff
6600 38       -          sec
6601 e5 8d    -          sbc  yoff        ;y, dann kein
6603 90 06    -          bcc  lq12        ;punkt setzen
6605 aa       -          tax              ;y-koordinate
6606 a4 fa    -          ldy  xl          ;x-koordinate low
6608 20 9c 60-           jsr  plot        ;punkt setzen
660b e6 ff    -lq12      inc  zaehl+1
660d a5 ff    -          lda  zaehl+1
660f c5 fe    -          cmp  zaehl       ;alle punkte
6611 d0 c5    -          bne  loopq1      ;gesetzt, dann
6613 60       -          rts              ;fertig
              -;
              -;
              -;zweiten quadranten zeichnen
              -;===========================
              -;
6614 85 fe    -zweiter   sta  zaehl       ;zaehler setzen
6616 86 ff    -          stx  zaehl+1
6618 a6 fe    -loopq2    ldx  zaehl       ;zaehler holen
661a bd 00 d0-           lda  $d000,x     ;x-odffset holen
661d 85 8c    -          sta  xoff        ;merken
661f bd 00 d1-           lda  $d100,x     ;y-offset holen
6622 85 8d    -          sta  yoff        ;merken
6624 a5 8e    -          lda  x1          ;mittelpunkts-
6626 a4 8f    -          ldy  x2          ;koordinaten holen
6628 18       -          clc
6629 65 8c    -          adc  xoff        ;x=x+xoff
662b 90 01    -          bcc  lq21
662d c8       -          iny
662e c0 00    -lq21      cpy  #00         ;xh=0, dann o.k.
6630 f0 08    -          beq  lq23
6632 c0 02    -          cpy  #02         ;xh1, dann kein
6634 b0 19    -          bcs  lq22        ;punkt setzen
6636 c9 40    -          cmp  #64         ;xh=1 und xl=64,
```

```
6638 b0 15   -          bcs  lq22        ;dann kein punkt
663a 85 fa   -lq23      sta  xl
663c 84 fb   -          sty  xh
663e a5 fc   -          lda  y           ;y=y-yoff
6640 18      -          clc
6641 65 8d   -          adc  yoff        ;y200, dann kein
6643 b0 0a   -          bcs  lq22
6645 c9 c8   -          cmp  #200        ;punkt setzen
6647 b0 06   -          bcs  lq22
6649 aa      -          tax              ;y-koordinate
664a a4 fa   -          ldy  xl          ;x-koordinate low
664c 20 9c 60-          jsr  plot        ;punkt setzen
664f e6 fe   -lq22      inc  zaehl
6651 a5 fe   -          lda  zaehl
6653 c5 ff   -          cmp  zaehl+1     ;alle punkte
6655 d0 c1   -          bne  loopq2      ;gesetzt, dann
6657 60      -          rts              ;fertig
             -;
             -;
             -;dritten quadranten zeichnen
             -;===========================
             -;
6658 85 fe   -dritter   sta  zaehl
665a 86 ff   -          stx  zaehl+1
665c a9 c3   -          lda  #195        ;zaehler setzen
665e 38      -          sec
665f e5 fe   -          sbc  zaehl       ;und invertieren
6661 85 fe   -          sta  zaehl
6663 a9 c3   -          lda  #195
6665 38      -          sec
6666 e5 ff   -          sbc  zaehl+1
6668 85 ff   -          sta  zaehl+1
666a a6 ff   -loopq3    ldx  zaehl+1     ;zaehler holen
666c bd 00 d0-          lda  $d000,x     ;x-odffset holen
666f 85 8c   -          sta  xoff        ;merken
6671 bd 00 d1-          lda  $d100,x     ;y-offset holen
6674 85 8d   -          sta  yoff        ;merken
6676 a5 8e   -          lda  x1          ;mittelpunkts-
6678 a4 8f   -          ldy  x2          ;koordinaten holen
667a 38      -          sec
667b e5 8c   -          sbc  xoff
667d b0 01   -          bcs  lq35        ;x=x-xoff
667f 88      -          dey
6680 c0 00   -lq35      cpy  #00         ;xh=0, dann o.k.
6682 f0 08   -          beq  lq36
6684 c0 02   -          cpy  #02         ;xh1, dann kein
```

```
6686 b0 19    -          bcs  lq38       ;punkt setzen
6688 c9 40    -          cmp  #64        ;xh=1 und xl=64,
668a b0 15    -          bcs  lq38       ;dann kein punkt
668c 85 fa    -lq36      sta  xl
668e 84 fb    -          sty  xh
6690 a5 fc    -          lda  y          ;y=y+yoff
6692 18       -          clc
6693 65 8d    -          adc  yoff
6695 b0 0a    -          bcs  lq38
6697 c9 c8    -          cmp  #200       ;y=200, dann kein
6699 b0 06    -          bcs  lq38       ;punkt setzen
669b aa       -          tax             ;y-koordinate
669c a4 fa    -          ldy  xl         ;x-koordinate low
669e 20 9c 60-           jsr  plot       ;punkt setzen
66a1 e6 ff    -lq38      inc  zaehl+1
66a3 a5 ff    -          lda  zaehl+1
66a5 c5 fe    -          cmp  zaehl      ;alle punkte
66a7 d0 c1    -          bne  loopq3     ;gesetzt, dann
66a9 60       -          rts             ;fertig
              -;
              -;
              -;vierten quadranten zeichnen
              -;===========================
              -;
66aa 85 fe    -vierter   sta  zaehl      ;zaehler setzen
66ac 86 ff    -          stx  zaehl+1
66ae a6 fe    -loopq4    ldx  zaehl      ;zaehler holen
66b0 bd 00 d0-           lda  $d000,x    ;x-odffset holen
66b3 85 8c    -          sta  xoff       ;merken
66b5 bd 00 d1-           lda  $d100,x    ;y-offset holen
66b8 85 8d    -          sta  yoff       ;merken
66ba a5 8e    -          lda  x1         ;mittelpunkts-
66bc a4 8f    -          ldy  x2         ;koordinaten holen
66be 38       -          sec
66bf e5 8c    -          sbc  xoff
66c1 b0 01    -          bcs  lq45       ;x=x-xoff
66c3 88       -          dey
66c4 c0 00    -lq45      cpy  #00        ;xh=0, dann o.k.
66c6 f0 08    -          beq  lq46
66c8 c0 02    -          cpy  #02        ;xhl, dann kein
66ca b0 15    -          bcs  lq48       ;punkt setzen
66cc c9 40    -          cmp  #64        ;xh=1 und xl=64,
66ce b0 11    -          bcs  lq48       ;dann kein punkt
66d0 85 fa    -lq46      sta  xl
66d2 84 fb    -          sty  xh
66d4 a5 fc    -          lda  y          ;y=y-yoff
```

```
66d6 38       -          sec
66d7 e5 8d    -          sbc  yoff       ;y, dann kein
66d9 90 c6    -          bcc  lq38       ;punkt setzen
66db aa       -          tax             ;y-koordinate
66dc a4 fa    -          ldy  xl         ;x-koordinate low
66de 20 9c 60-           jsr  plot       ;punkt setzen
66e1 e6 fe    -lq48      inc  zaehl
66e3 a5 fe    -          lda  zaehl
66e5 c5 ff    -          cmp  zaehl+1    ;alle punkte
66e7 d0 c5    -          bne  loopq4     ;gesetzt, dann
66e9 60       -          rts             ;fertig
              -;
              -;koordinaten holen
              -;=================
              -;
66ea 20 58 63-koor       jsr  $6358      ;mittelpunkt holen
66ed 20 fd ae-           jsr  komma
66f0 20 9e b7-           jsr  byte       ;x-radius holen
66f3 e0 81    -          cpx  #129       ;= 129
66f5 b0 5a    -          bcs  error      ;ja, fehler
66f7 86 fd    -          stx  xr
66f9 20 fd ae-           jsr  komma
66fc 20 9e b7-           jsr  byte       ;y-radius holen
66ff e0 81    -          cpx  #129       ;= 129
6701 b0 4e    -          bcs  error      ;ja, fehler
6703 86 8b    -          stx  yr
6705 20 fd ae-           jsr  komma
6708 20 1a 67-           jsr  rechne     ;anfangswinkel in
670b 85 b4    -          sta  anqua      ;quadranten und
670d 84 b6    -          sty  anpunkt    ;punkt umrechnen
670f 20 fd ae-           jsr  komma
6712 20 1a 67-           jsr  rechne     ;endwinkel in
6715 85 b5    -          sta  endqua     ;quadranten und
6717 84 72    -          sty  endpunkt   ;punkt umrechnen
6719 60       -          rts
              -;
              -;umrechnung winkel-quadranten/punkt
              -;==================================
              -;
671a 20 8a ad-rechne     jsr  frmnum     ;ausdruck holen
671d a2 7f    -          ldx  #(buffer)
671f a0 67    -          ldy  #(buffer)
6721 20 d4 bb-           jsr  facmem     ;wert sichern
6724 a9 59    -          lda  #(multtabl)
6726 a0 67    -          ldy  #(multtabl)
6728 20 28 ba-           jsr  memmult    ;fac=fac*(4/2*=)
```

```
6672b 20 aa b1-          jsr  intfac      ;a/y=int(fac)
672e c8       -          iny
672f 98       -          tya              ;plus 1 =
6730 48       -          pha
6731 aa       -          tax
6732 ca       -          dex              ;quadrant merken
6733 bd 67 67-           lda  quatabh,x
6736 a8       -          tay              ;fac mit
6737 bd 63 67-           lda  quatabl,x   ;(quadrant-1)*
673a 20 a2 bb-           jsr  memfac      ;(=/2) laden
673d a9 7f    -          lda  #(buffer)
673f a0 67    -          ldy  #(buffer)
6741 20 50 b8-           jsr  memmin      ;fac=wert-fac
6744 a9 5e    -          lda  #(multtab2)
6746 a0 67    -          ldy  #(multtab2)
6748 20 28 ba-           jsr  memmult     ;fac=fac*(195*2/=)
674b 20 aa b1-           jsr  intfac      ;a/y=int(fac)
674e c8       -          iny
674f 68       -          pla
6750 60       -          rts              ;zahl
              -;
6751 68       -error     pla
6752 68       -          pla
6753 20 18 60-           jsr  groff       ;grafik aus
6756 4c 48 b2-           jmp  $b248       ;illegal quantity
              -;
6759 80 22 f9-multtabl  .by 128,34,249,131,109
675e 87 78 48-multtab2  .by 135,120,72,30,56
6763 6b 70 75-quatabl   .by(eins),(zwei),(drei),(vier)
6767 67 67 67-quatabh   .by(eins),(zwei),(drei),(vier)
676b 00 00 00-eins      .by 0,0,0,0,0
6770 81 49 0f-zwei      .by 129,73,15,218,162
6775 82 49 0f-drei      .by 130,73,15,218,162
677a 83 16 cb-vier      .by 131,22,203,227,250
677f 00 00 00-buffer    .by 0,0,0,0,0,0
              -;
              -;berechnung von akku*x/128
              -;========================
              -;
6785 85 71    -mult      sta  mu
6787 8e 93 67-           stx  mult1+1
678a a2 07    -          ldx  #07
678c a9 00    -          lda  #00
678e 46 71    -m1        lsr  mu
6790 90 02    -          bcc  m2
6792 69 00    -mult1     adc  #00
```

```
705:    6794 6a       -m2        ror  a
706:    6795 ca       -          dex
707:    6796 d0 f6    -          bne  m1
708:    6798 60       -          rts
709:                  -;
710:                  -;cosinus-sinus-tabelle
711:                  -;=====================
712:                  -;
713:    6799 00 01 02-cos     .by0,1,2,3,4,5,6,7,8,9,10
714:    67a4 0b 0c 0d-.by11,12,13,14,15,16,17,18,19,20
715:    67ae 15 16 17-.by21,22,23,24,25,26,27,28,29,30
716:    67b8 1f 20 21-.by31,32,33,34,35,36,37,38,39,40
717:    67c2 29 2a 2b-.by41,42,43,44,45,46,47,48,49,50
718:    67cc 33 34 35-.by51,52,53,53,54,55,56,57,58,59
719:    67d6 3c 3d 3e-.by60,61,62,63,64,64,65,66,67,68
720:    67e0 45 46 47-.by69,70,71,71,72,73,74,75,76,76
721:    67ea 4d 4e 4f-.by77,78,79,80,80,81,82,83,84,84
722:    67f4 55 56 57-.by85,86,87,87,88,89,90,90,91,92
723:    67fe 5d 5d 5e-.by93,93,94,95,95,96,97,97,98,99
724:    6808 63 64 65-.by099,100,101,101,102,102,103,104,104,105
725:    6812 69 6a 6b-.by105,106,107,107,108,108,109,109,110,110
726:    681c 6f 6f 70-.by111,111,112,112,113,113,114,114,115,115
727:    6826 74 74 74-.by116,116,116,117,117,118,118,118,119,119
728:    6830 78 78 78-.by120,120,120,121,121,121,122,122,122,122
729:    683a 7b 7b 7b-.by123,123,123,124,124,124,124,124,125,125
730:    6844 7d 7d 7d-.by125,125,125,126,126,126,126,126,126,127
731:    684e 7f 7f 7f-.by127,127,127,127,127,127,127,127,127,127
732:    6858 7f 7f 7f-.by127,127,127,127
```

Zu diesem Mammutlisting möchte ich abschließend noch einige Worte sagen. Zunächst zu der Umrechnung des Winkels in den Quadranten und den Startpunkt (Zeilen 653–678). Wenn Sie als Anwender diese Routine benutzen wollen, möchten Sie natürlich den Start- und Endwinkel gern im Bogenmaß eingeben, womit das Programm allerdings nicht viel anfangen kann. Daher muß der Winkel umgerechnet werden. Zunächst ist der Quadrant zu bestimmen. Hierfür ist folgende Rechnung erforderlich:

```
Quadrant = INT(Winkel*(4/(2*PI)))+1
```

Der Winkel muß dabei zwischen 0 und 1,999*PI liegen. Wenn Sie z.B. als Winkel 100 Grad (=0,8726*PI) eingeben, wird nach der Formel INT(0,8726*4/2)+1 der 2.Quadrant berechnet, was sicherlich richtig ist, da dieser von 90–180 Grad geht. Nun muß noch der Wert innerhalb des Quadranten gefunden werden, bei dem die Schleife (s.o.) zu starten hat. Hier können Werte von 0 bis 195 berechnet werden nach folgender Formel:

```
Wert = INT(Winkel-((Quadrant-1)*(PI/2))*(195/(PI/2)))
```

Diese kompliziert erscheinende Formel ist jedoch durchaus erklärbar: Um das Verhältnis des Winkels zu 90 Grad zu finden, muß zunächst der Gesamtwinkel auf einen Anteil reduziert werden, der zwischen 0 und 90 Grad liegt. Dafür wird 90 Grad (=PI/2) einmal weniger abgezogen als der Wert des Quadranten, z.B. bei 100 Grad (2. Quadrant) werden einmal 90 Grad subtrahiert, so daß das Ergebnis 10 Grad lautet. Dies ist der Anteil des Winkels, der seit Eintritt in den Quadranten (bei 90 Grad) überstrichen worden ist. Nun wird dieses Ergebnis noch mit dem Faktor 195/(PI/2) multipliziert, um den Winkel in den Startpunkt umzurechnen. Der Faktor ergibt sich aus dem möglichen Punktwert (0–195) und dem überstrichenen Winkel eines Quadranten (PI/2). Bei dem Zwischenergebnis von 10 Grad hieße es, daß der Startpunkt unserer Ellipse bei INT(0.027777*PI(=10Grad)*195/(PI/2))= 10 zu finden ist. Genau dasselbe Verfahren wird angewandt, wenn es darum geht, den Endpunkt umzurechnen. Falls man eine Vollellipse wünscht, muß man als Startpunkt 0 und als Endpunkt 1.999*PI eingeben. In den Zeilen 136–148 wird überprüft, ob eine ganze Ellipse oder nur ein Ausschnitt gezeichnet werden soll. Die Routine für die Komplett-Ellipse befindet sich in den Zeilen 152–247 und arbeitet nach dem oben beschriebenen Verfahren. Sehr viel komplizierter ist es, einen Ausschnitt zu berechnen. Es gibt nämlich 16 Möglichkeiten der Quadrantenkombination:

Startquadrant	Endquadrant
1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
3	1
3	2
3	3
3	4
4	1
4	2
4	3
4	4

Für jede Möglichkeit mußte eine eigene Routine geschrieben werden, die in Zeile 293 angesprungen wird. Die Startadresse dieser Routine wird aus einer Tabelle ausgelesen. Der Index zum Lesen der Tabelle wird in den Zeilen 283–288 berechnet, einfach, indem man den Anfangsquadranten mal vier nimmt und den Endquadranten hinzuaddiert. Dadurch sind Werte von 5 bis 20 möglich, wobei jeder Wert eindeutig einer Quadrantenkombination zugeordnet ist. Als Beispiel möchte ich die Routine besprechen, die den Ausschnitt vom ersten in den

dritten Quadranten zeichnet (Zeilen 323–331): Zunächst muß der erste Quadrant betrachtet werden. Hier ist genau der Abschnitt vom Start (ANPUNKT) bis zum Ende (195) zu zeichnen. Der zweite Quadrant muß komplett aufgebaut werden (Start:1, Ende:195), während der dritte Quadrant genau vom Anfang (1) bis zum berechneten Endpunkt gezeichnet werden muß. Die folgenden Programmzeilen bis 442 behandeln auf diese Weise auch die übrigen 15 Kombinationen.

Ab Zeile 454 beginnt der Programmteil, der für den Aufbau der einzelnen Quadranten zuständig ist. Er funktioniert genauso wie bei der Vollellipse, hier wird jedoch immer nur ein Punkt in dem zuständigen Quadranten gezeichnet und nicht vier Punkte in allen Quadranten.

Die Routine ab Zeile 698 dient dazu, die Berechnung von XR*Wert(I)/128 durchzuführen. Die Ellipsenroutine gehört zu den schnellsten jemals für den C64 verfaßten, bei maximaler Genauigkeit. Sie zeichnet mehrere Kreise/Ellipsen pro Sekunde, so daß der Aufbau eines einzelnen nicht mit dem Auge verfolgt werden kann. Dies ist der Punkt, der eine »schnelle« von einer »langsamen« Routine unterscheidet. Aufgerufen wird sie mit

```
SYS 25483,XM,YM,M,XR,YR,AW,EW
```

XM: Mittelpunkt der Ellipse in X-Richtung
YM: Mittelpunkt der Ellipse in Y-Richtung
M: Zeichenmodus (32=löschen, 64=setzen, 128=invertieren)
XR: X-Radius (muß kleiner oder gleich 128 sein)
YR: Y-Radius (muß kleiner oder gleich 128 sein)
AW: Anfangswinkel im Bogenmaß (zwischen 0 und 1,999*PI)
EW: Endwinkel im Bogenmaß (zwischen 0 und 1,999*PI)

4.5 Das Schreiben von Text in die HiRes-Grafik

Für die Realisierung des Text-Befehls gibt es prinzipiell zwei Möglichkeiten: Zum einen gibt es Erweiterungen, die durch die Interrupttechnik zwischen Text- und Grafikschirm hin- und herschalten. Der Betrachter hat durch die ungeheure Geschwindigkeit das Gefühl, beide Schirme gleichzeitig zu sehen. Der Vorteil dieser Methode besteht darin, daß man den normalen Textschirm zum Erstellen der Texte benutzen kann, also den PRINT-Befehl mit allen Steuerzeichen. Der Nachteil wiegt jedoch schwer: Da die Schriftzeichen nicht im Grafikspeicher vorhanden sind, können sie bei einem Grafikausdruck nicht berücksichtigt werden und werden auch bei einer Speicherung der Grafik auf Diskette übergangen. Man müßte praktisch immer auch den Text-Schirm mitausgeben. Ein weiterer Nachteil ist, daß die Textzeichen nicht beliebig, sondern nur an 1000 Positionen ausgegeben werden können, da der Textschirm ja nur die Koordinaten 0–39 in x-Richtung und 0–24 in y-Richtung aufweist. Dies alles führte dazu, daß sich diese Methode nicht durchsetzen konnte.

Die zweite Möglichkeit besteht darin, den Zeichengenerator ab $D000 auszulesen und die Informationen direkt in den Grafikspeicher zu schreiben. Damit sind die oben geschilderten Nachteile behoben. Man hat weiterhin die Möglichkeit, die Texte in Größe und Aussehen zu manipulieren, wovon wir in unserer Routine reichlich Gebrauch machen werden. Zunächst soll jedoch auf den Aufbau des Zeichengenerators eingegangen werden.

Dieser belegt den Speicherraum von $D000-$DFFF, wobei der Teil bis $D7FF für den Klein-/Großschriftmodus zuständig ist, der Bereich ab $D800 für den Großschrift/Grafikmodus. Bei 256 möglichen Zeichen belegt also jedes einzelne genau 8 Bytes,wobei sich die Reihenfolge nach dem Bildschirmcode richtet. Das erste Zeichen ist also der Klammeraffe (Code 0), welcher den Bereich von $D000 bis $D007 belegt. Es folgt das »A« von $D008-$D00F u.s.w. Der eigentliche Aufbau der Zeichen ist analog zum Grafikspeicher zu sehen: Die 8*8-Matrix jedes Zeichens wird durch 64 Bits dargestellt, wobei jedes gesetzte Bit bedeutet, daß der Punkt in der Zeichenmatrix gesetzt wird und umgedreht. Jede Zeile der Zeichenmatrix wird durch ein Byte repräsentiert, was wir an Hand eines Beispiels verdeutlichen wollen. Das Zeichen »*«, das den Bildschirmcode 42 aufweist, belegt daher die Bytes $D150 bis $D157. Im folgenden soll ein »*« einen gesetzten Punkt bedeuten, ein ».« einen nicht gesetzten:

Adresse	**Inhalt**	**Bits**
		76543210
$D150	0	....
$D151	102	.**..**.
$D152	60	..****.
$D153	255	********
$D154	60	..****..
$D155	102	.**..**.
$D156	0	....
$D157	0	....

Auch hier erkennt man wie beim Grafikspeicher einen 8*8-Block, wobei die Wertigkeit der Bits von links nach rechts reicht. Wenn wir also ein Zeichen auf dem Grafikschirm darstellen wollen, müssen wir die 64 zugehörigen Bits aus dem Zeichengenerator auslesen und bei gesetztem Bit einen Grafikpunkt setzen.

Nun wollen wir zu der Möglichkeit übergehen, den Text in Größe und Aussehen zu manipulieren. Zunächst wollen wir die Größe behandeln. Wenn wir das Zeichen in X-Richtung vergrößern oder verkleinern wollen, müssen wir Grafikpunkte entweder mehrfach oder überhaupt nicht setzen, wie wir an dem folgenden Beispiel sehen. Zunächst soll der »*« in x-Richtung um den Faktor 2 gestreckt werden:

```
76543210          7766554433221100
........          ................
.**..**.          ..****....****..
..****..          ....********....
******** x*2 =    ****************
..****..          ....********....
.**..**.          ..****....****..
........          ................
........          ................
```

Man sieht, daß jeder Punkt doppelt gesetzt wurde. Auch gebrochene Faktoren stellen kein Problem dar, so muß z.B. beim Faktor 1,5 nur jeder zweite Punkt doppelt gesetzt werden, während die übrige Hälfte nur einfach gezeichnet wird:

```
76543210           766544322100
.**..**.           .***...***..
..****..           ...******...
******** x*1,5 =   ************
..****..           ...******...
.**..**.           .***...***..
........           ............
........           ............
```

Nun kommen wir zu einer Verkleinerung um den Faktor 0,5. Hier muß deshalb jeder zweite Punkt unterdrückt werden:

```
76543210           7531
........           ....
.**..**.           .*.*
..****..           .**.
******** x*0,5 =   ****
..****..           .**.
.**..**.           .*.*
........           ....
........           ....
```

Leider muß man bei Verkleinerungen, wie hier ersichtlich, mit Verkrüppelungen des Zeichens rechnen. Die besten Ergebnisse in dieser Hinsicht zeigen völlig regelmäßig aufgebaute Zeichen. Im allgemeinen werden verkleinerte Texte jedoch so unleserlich, daß man sie am besten in der original 8*8-Größe beläßt.

Die Vergrößerung/Verkleinerung in Y-Richtung funktioniert ähnlich. Hier müssen jedoch nicht Bits, sondern ganze Zeichensatzbytes bearbeitet werden. Es leuchtet z.B. unmittelbar ein, daß bei einer Vergrößerung um den Faktor 1,25 jede vierte Zeile (= jedes vierte Byte) doppelt dargestellt werden muß:

```
  76543210           76543210
0 ........         0 ........
1 .**..**.         1 .**..**.
2 ..****..         2 ..****..
3 ******** y*1,25= 3 ********
4 ..****..         3 ********
5 .**..**.         4 ..****..
6 ........         5 .**..**.
7 ........         6 ........
                   7 ........
                   7 ........
```

Analog müssen bei Verkleinerungen ganze Zeilen (Bytes) unterdrückt werden. Natürlich kann man nun Vergrößerungen oder Verkleinerungen beliebig kombinieren, wie wir es nachher im Text-Befehl auch praktizieren werden.

Der zweite Punkt betrifft das Aussehen eines Zeichens. Hiermit meine ich eine eventuelle Rotation oder eine kursive Verzerrung. Beide Effekte lassen sich dadurch erreichen, daß man die beiden für das Aussehen zuständigen Vektoren abändert. Diese beiden Vektoren sind

S: der Vektor von einem Punkt zum nächsten innerhalb einer Zeile
Z: der Vektor von einem Zeilenanfang zum nächsten

Im Normalfall erhält man den nächsten Punkt in einer Zeile dadurch, daß man die X-Koordinate um eins erhöht und die Y-Koordinate gleich läßt. Von einer Zeile zur nächsten gelangt man, indem man sich einen Punkt in y-Richtung bewegt und die X-Koordinate gleichläßt. Die Vektoren haben also die Inhalte

S(X,Y) = (1,0) und Z(X,Y) = (0,1)

Wie kann man nun z.B. eine rechtskursive Verzerrung erhalten? Nun, man ändert die Werte des Zeilenvektors wie folgt ab:

Z(X,Y) = (–1,1)

Damit wird der Startpunkt der folgenden Zeile immer um einen Punkt weiter nach links verschoben, wodurch der Kursiv-Effekt eintritt:

```
0 ........                  0 ........
1 .**..**.                 1 .**..**.
2 ..****..                2 ..****..
3 ******** Z=(-1,1)      3 ********
4 ..****..              4 ..****..
5 .**..**.             5 .**..**.
6 ........            6 ........
7 ........           7 ........
```

Eine Rotation kann man durch Abänderung des Spalten- und des Zeilenvektors erreichen. Wir wollen nun z.B. den Stern um 90 Grad nach rechts kippen. Der Spaltenvektor muß nun dafür

sorgen, daß die Punkte innerhalb einer Zeile nicht mehr nebeneinander, sondern übereinander liegen, der Zeilenvektor dafür, daß die Zeilen in x-Richtung fortschreiten:

```
0 ........           ...*....
1 .**..**.           .*.*.*..
2 ..****..           .*****..
3 ******** Z=(1,0)   ..***...
4 ..****.. S=(0,-1)  ..***...
5 .**..**.           .*****..
6 ........           .*.*.*..
7 ........           ...*....
                     01234567
```

Natürlich kann man auch die Effekte Rotation und Kursivverzerrung kombinieren. Die folgende Tabelle zeigt den Zusammenhang zwischen den wichtigsten Spalten- sowie Zeilenvektoren und den Effekten Rotation, Verzerrung und Spiegelung:

Zeilenvektor		Spaltenvektor		Rotation (Grad)	Verzerrung (kursiv)	Spiegelung (Achse)
x	y	x	y			
0	1	1	0	0	–	–
0	1	–1	0	0	–	Y
0	–1	1	0	0	–	X
0	–1	–1	0	0	–	X,Y
–1	1	1	0	0	links	–
1	1	1	0	0	rechts	–
1	1	1	–1	45	–	–
1	0	0	–1	90	–	–
1	–1	0	–1	90	links	–
1	1	0	–1	90	rechts	–
1	–1	–1	–1	135	–	–
0	–1	–1	0	180	–	–
–1	–1	–1	0	180	links	–
1	–1	–1	0	180	rechts	–
–1	–1	–1	1	225	–	–
–1	0	0	1	270	–	–
–1	–1	0	1	270	links	–
–1	1	0	1	270	rechts	–
–1	1	1	1	315	–	–

Durch Verknüpfung mit der Variation der Größe kann man einem Zeichen praktisch jedes beliebige Aussehen verleihen. Der nachfolgende Assemblerquelltext beinhaltet den vielleicht mächtigsten Text-Befehl seit Einführung des C64 im Jahr 1982. Durch die ungeheuer komplizierten Berechnungen kann der Befehl allerdings keine Geschwindigkeitsrekorde brechen. Falls Sie einen schnelleren Befehl benötigen, müssen Sie auf einen einfacheren Algorithmus ausweichen.

Listing: »text«

```
6a4e          -;text
6a4e          -           .ba $6a4e
              -;
b79e          -           .eq byte      = $b79e ;byte holen
aefd          -           .eq komma     = $aefd ;komma
ad9e          -           .eq frmevl    = $ad9e ;ausdruck
b1bf          -           .eq integer   = $b1bf ;nach int.
00fa          -           .eq xsl       = $fa   ;spaltenkoor-
00fb          -           .eq xsh       = $fb   ;dinate in
00fc          -           .eq ys        = $fc   ;einer zeile
008b          -           .eq xzl       = $8b   ;koordin. vom
008c          -           .eq xzh       = $8c   ;zeilenanfang
008d          -           .eq yz        = $8d   ;im zeichen
0095          -           .eq xl        = $95   ;linke, obere
0096          -           .eq xh        = $96   ;ecke=bezugs-
008e          -           .eq y         = $8e   ;koordinate
008f          -           .eq xabzl     = $8f   ;zeilenab-
00fd          -           .eq xabzh     = $fd   ;stand im
00fe          -           .eq yabz      = $fe   ;zeichen
00b5          -           .eq xabsl     = $b5   ;spaltenab-
00b6          -           .eq xabsh     = $b6   ;stand in
00b7          -           .eq yabs      = $b7   ;einer zeile
00b8          -           .eq xabl      = $b8   ;zeichenab-
00b9          -           .eq xabh      = $b9   ;stand in
00ba          -           .eq yab       = $ba   ;einem string
00bb          -           .eq xfvk      = $bb   ;faktoren der
00bc          -           .eq xfnk      = $bc   ;vergrößerung
00bd          -           .eq yfvk      = $bd   ;vor-und nach
00be          -           .eq yfnk      = $be   ;kommaanteile
00bf          -           .eq xvk       = $bf   ;faktoren,die
00b4          -           .eq xnk       = $b4   ;haeufigkeit
0010          -           .eq yvk       = $10   ;der bits
0002          -           .eq ynk       = $02   ;bestimmen
033c          -           .eq zaehler   = $033c ;8 bytes
033d          -           .eq laenge    = $033d ;stringlaenge
033e          -           .eq posi      = $033e ;stringposit.
033f          -           .eq zeichen   = $033f ;zeichencode
009c          -           .eq revers    = $9c   ;reversflag
0092          -           .eq rechen1   = $92   ;rechen-
0093          -           .eq rechen2   = $93   ;register
0340          -           .eq satz      = $0340 ;zeichensatz
6088          -           .eq plot      = $6088 ;setz punkt
              -;
6a4e 20 fd ae-start       jsr  komma
6a51 20 58 63-            jsr  $6358     ;startkoordinaten
```

```
6a54 a5 fa    -    lda  xsl      ;holen und
6a56 a6 fb    -    ldx  xsh
6a58 a4 fc    -    ldy  ys       ;setzen
6a5a 85 95    -    sta  xl
6a5c 86 96    -    stx  xh       ;(linke, obere ecke
6a5e 84 8e    -    sty  y        ;des ersten zeichen
6a60 20 fd ae-     jsr  komma
6a63 20 9e ad-     jsr  frmevl   ;zeiger auf string-
6a66 a0 00    -    ldy  #00      ;descriptor holen
6a68 b1 64    -    lda  ($64),y
6a6a 8d 3d 03-     sta  laenge   ;laenge holen
6a6d c8       -    iny
6a6e b1 64    -    lda  ($64),y
6a70 48       -    pha           ;stringzeiger holen
6a71 c8       -    iny
6a72 b1 64    -    lda  ($64),y  ;und sichern
6a74 48       -    pha
6a75 20 48 6c-     jsr  holpar   ;vektor von einem
6a78 a5 64    -    lda  $64
6a7a a4 65    -    ldy  $65
6a7c 84 8f    -    sty  xabzl    ;zeilenanfang zum
6a7e 85 fd    -    sta  xabzh    ;naechsten (in x)
6a80 20 48 6c-     jsr  holpar
6a83 a4 65    -    ldy  $65
6a85 84 fe    -    sty  yabz     ;(in y)
6a87 20 48 6c-     jsr  holpar   ;vektor von einem
6a8a a5 64    -    lda  $64
6a8c a4 65    -    ldy  $65
6a8e 84 b5    -    sty  xabsl    ;punkt innerhalb
6a90 85 b6    -    sta  xabsh    ;einer zeile zum
6a92 20 48 6c-     jsr  holpar   ;naechsten in
6a95 a4 65    -    ldy  $65
6a97 84 b7    -    sty  yabs     ;x- und y-richtung
6a99 20 51 6c-     jsr  holbyte  ;vergroesserungs-
6a9c 86 bb    -    stx  xfvk     ;faktoren holen
6a9e 20 51 6c-     jsr  holbyte  ;x-und y-
6aa1 86 bc    -    stx  xfnk     ;jeweils vorkomma-
6aa3 20 51 6c-     jsr  holbyte  ;und nachkomma-
6aa6 86 bd    -    stx  yfvk     ;anteil
6aa8 20 51 6c-     jsr  holbyte
6aab 86 be    -    stx  yfnk
6aad 20 48 6c-     jsr  holpar   ;abstand von einem
6ab0 a5 64    -    lda  $64
6ab2 a4 65    -    ldy  $65      ;zeichen zum
6ab4 84 b8    -    sty  xabl
6ab6 85 b9    -    sta  xabh     ;naechsten holen
```

```
6ab8 20 48 6c-          jsr  holpar
6abb a4 65    -         ldy  $65
6abd 84 ba    -         sty  yab
6abf 68       -         pla
6ac0 85 65    -         sta  $65
6ac2 68       -         pla
6ac3 85 64    -         sta  $64
              -;
              -;textauswertung-start
              -;===================
              -;
6ac5 78       -         sei
6ac6 a9 34    -         lda  #52       ;prozessorport auf
6ac8 85 01    -         sta  $01       ;ram schalten
6aca a9 d0    -         lda  #$d0      ;zeichensatzstart
6acc 8d 40 03-          sta  satz      ;ab $d000
6acf a9 00    -         lda  #00
6ad1 8d 3e 03-          sta  posi      ;position im string
6ad4 85 9c    -         sta  revers    ;reversflag loeschen
6ad6 ac 3e 03-loop      ldy  posi
6ad9 cc 3d 03-          cpy  laenge    ;position=laenge
6adc f0 6a    -         beq  out       ;ja, fertig
6ade b1 64    -         lda  ($64),y   ;zeichen holen
6ae0 aa       -         tax
6ae1 29 7f    -         and  #$7f      ;ascii-code <_32,
6ae3 c9 20    -         cmp  #32
6ae5 90 17    -         bcc  sonder    ;dann sonderzeichen
6ae7 8a       -         txa
6ae8 29 80    -         and  #$80      ;ascii- in
6aea 4a       -         lsr  a
6aeb 85 92    -         sta  rechen1   ;schirmcode
6aed 8a       -         txa
6aee 29 3f    -         and  #$3f      ;umwandeln
6af0 05 92    -         ora  rechen1
6af2 20 52 6b-          jsr  text      ;zeichen ausgeben
6af5 20 33 6c-          jsr  nextzeich ;naechsten start-
6af8 ee 3e 03-wei       inc  posi      ;punkt berechnen +
6afb 4c d6 6a-          jmp  loop      ;position erhoehen
              -;
              -;sonderzeichen bearbeiten
              -;========================
              -;
6afe e0 93    -sonder   cpx  #$93
6b00 f0 17    -         beq  clr       ;bildschirm loeschen
6b02 e0 13    -         cpx  #$13
6b04 f0 22    -         beq  home      ;cursor home
```

```
6b06 e0 12   -          cpx  #$12
6b08 f0 29   -          beq  rvson   ;text revers
6b0a e0 92   -          cpx  #$92
6b0c f0 28   -          beq  rvsoff  ;text normal
6b0e e0 01   -          cpx  #$01
6b10 f0 2b   -          beq  klein   ;klein/grosschrift
6b12 e0 02   -          cpx  #$02
6b14 f0 2a   -          beq  gross   ;gross/grafikmodus
6b16 4c f8 6a-          jmp  wei     ;undefinierbar
             -;
6b19 a5 fa   -clr       lda  xsl     ;koordinaten retten
6b1b 48      -          pha
6b1c a5 fb   -          lda  xsh
6b1e 48      -          pha
6b1f 20 6e 60-          jsr  $606e   ;bildschirm loeschen
6b22 68      -          pla
6b23 85 fb   -          sta  xsh     ;koordinaten wieder
6b25 68      -          pla
6b26 85 fa   -          sta  xsl     ;setzen
             -;
6b28 a9 00   -home      lda  #00     ;x- und y-
6b2a 85 95   -          sta  xl      ;koordinaten auf
6b2c 85 96   -          sta  xh      ;null setzen
6b2e 85 8e   -          sta  y
6b30 4c f8 6a-          jmp  wei
             -;
6b33 a9 80   -rvson     lda  #$80    ;reversflag setzen
6b35 2c      -          .byte$2c
6b36 a9 00   -rvsoff    lda  #00     ;reversflag loeschen
6b38 85 9c   -          sta  revers
6b3a 4c f8 6a-          jmp  wei
             -;
6b3d a9 d8   -klein     lda  #$d8    ;start klein/gross
6b3f 2c      -          .byte$2c     ;zeichensatz
6b40 a9 d0   -gross     lda  #$d0    ;start gross/grafik
6b42 8d 40 03-          sta  satz    ;zeichensatz
6b45 4c f8 6a-          jmp  wei
             -;
6b48 a9 37   -out       lda  #55     ;prozessorport
6b4a 85 01   -          sta  $01     ;auf rom schalten
6b4c 58      -          cli
6b4d a9 19   -          lda  #25     ;zeiger auf
6b4f 85 16   -          sta  $16     ;stringstack zurueck
6b51 60      -          rts          ;fertig
             -;
             -;textausgabe eines zeichens
```

```
                    -;=========================
                    -;
6b52 05 9c          -text      ora  revers   ;reversflag in
6b54 a2 00          -          ldx  #00      ;code einblenden
6b56 8e 3c 03-                 stx  zaehler  ;zaehler fuer bytes
6b59 86 93          -          stx  rechen2
6b5b 0a             -          asl  a        ;bildschirmcode
6b5c 26 93          -          rol  rechen2
6b5e 0a             -          asl  a        ;*8
6b5f 26 93          -          rol  rechen2
6b61 0a             -          asl  a        ;plus
6b62 26 93          -          rol  rechen2
6b64 85 92          -          sta  rechen1  ;high-byte vom
6b66 a5 93          -          lda  rechen2
6b68 6d 40 03-                 adc  satz     ;zeichensatz
6b6b 85 93          -          sta  rechen2  ;=startadresse
6b6d a5 bd          -          lda  yfvk
6b6f a4 be          -          ldy  yfnk     ;y-faktor
6b71 85 10          -          sta  yvk
6b73 84 02          -          sty  ynk      ;sowie
6b75 a5 95          -          lda  xl
6b77 a6 96          -          ldx  xh       ;startkoordinate
6b79 a4 8e          -          ldy  y
6b7b 85 fa          -          sta  xsl      ;uebernehmen
6b7d 86 fb          -          stx  xsh
6b7f 84 fc          -          sty  ys
6b81 ac 3c 03-                 ldy  zaehler
6b84 c6 01          -l1        dec  $01      ;auf zeichensatz
6b86 b1 92          -          lda  (rechen1),y ;byte holen
6b88 e6 01          -          inc  $01      ;auf ram schalten
6b8a 20 98 6b-                 jsr  wertaus  ;auswerten
6b8d ee 3c 03-                 inc  zaehler
6b90 ac 3c 03-                 ldy  zaehler  ;8 bytes pro
6b93 c0 08          -          cpy  #08
6b95 d0 ed          -          bne  l1       ;zeichen notwendig
6b97 60             -          rts
                    -;
                    -;byte eines zeichens auswerten
                    -;============================
                    -;
6b98 8d 3f 03-wertaus          sta  zeichen  ;code merken
6b9b a5 10          -loopw     lda  yvk      ;vorkommafaktor=0
6b9d f0 4e          -          beq  zaehly   ;ja, hochzaehlen
6b9f c6 10          -          dec  yvk
6ba1 a5 bb          -          lda  xfvk     ;x-faktor ueber-
6ba3 a6 bc          -          ldx  xfnk     ;nehmen
```

```
6ba5 85 bf    -          sta  xvk
6ba7 86 b4    -          stx  xnk
6ba9 a5 fa    -          lda  xsl     ;startkoordinaten
6bab a6 fb    -          ldx  xsh     ;der zeile ueber-
6bad a4 fc    -          ldy  ys      ;nehmen
6baf 85 8b    -          sta  xzl
6bb1 86 8c    -          stx  xzh
6bb3 84 8d    -          sty  yz
6bb5 ad 3f 03-           lda  zeichen
6bb8 48       -          pha
6bb9 a2 08    -          ldx  #08
6bbb 0e 3f 03-loopbit    asl  zeichen ;bit pruefen
6bbe 20 ce 6b-           jsr  pruefbit
6bc1 ca       -          dex          ;8 bits pro byte
6bc2 d0 f7    -          bne  loopbit ;erforderlich
6bc4 20 09 6c-           jsr  nextzei ;koordinaten der
6bc7 68       -          pla          ;naechsten zeile
6bc8 8d 3f 03-           sta  zeichen ;setzen
6bcb 4c 9b 6b-           jmp  loopw   ;naechste zeile
              -;
              -;prueft bits eines zeichen-bytes
              -;================================
              -;
6bce a9 00    -pruefbit  lda  #00     ;bit=0 oder 1
6bd0 2a       -          rol  a       ;in y-register
6bd1 a8       -          tay          ;schieben
6bd2 a5 bf    -loopp     lda  xvk     ;x-vorkommafaktor=0
6bd4 f0 25    -          beq  zaehlx  ;ja, hochzaehlen
6bd6 c6 bf    -          dec  xvk
6bd8 c0 00    -          cpy  #00     ;bit=0, kein punkt
6bda f0 0b    -          beq  noplot
6bdc 8a       -          txa
6bdd 48       -          pha          ;x- und y-register
6bde 98       -          tya          ;retten
6bdf 48       -          pha
6be0 20 88 60-           jsr  plot    ;punkt setzen
6be3 68       -          pla
6be4 a8       -          tay          ;x- und y-register
6be5 68       -          pla          ;wiederholen
6be6 aa       -          tax
6be7 20 1e 6c-noplot     jsr  nextspa ;koordinaten der
6bea 4c d2 6b-           jmp  loopp   ;naechsten spalte
              -;
6bed a5 02    -zaehly    lda  ynk     ;aktuelle
6bef 18       -          clc          ;y-faktoren =
6bf0 65 be    -          adc  yfnk
```

```
6bf2 85 02  -            sta  ynk     ;aktuelle y-faktoren
6bf4 a5 10  -            lda  yvk     ;plus
6bf6 65 bd  -            adc  yfvk    ;y-faktoren
6bf8 85 10  -            sta  yvk
6bfa 60     -            rts
            -;
6bfb a5 b4  -zaehlx      lda  xnk     ;aktuelle
6bfd 18     -            clc          ;x-faktoren =
6bfe 65 bc  -            adc  xfnk
6c00 85 b4  -            sta  xnk     ;aktuelle x-faktoren
6c02 a5 bf  -            lda  xvk     ;plus
6c04 65 bb  -            adc  xfvk    ;x-faktoren
6c06 85 bf  -            sta  xvk
6c08 60     -            rts
            -;
6c09 a5 8d  -nextzei     lda  yz      ;startkoordinaten
6c0b 18     -            clc
6c0c 65 fe  -            adc  yabz    ;der naechsten
6c0e 85 fc  -            sta  ys
6c10 a5 8b  -            lda  xzl     ;zeile setzen
6c12 18     -            clc
6c13 65 8f  -            adc  xabzl   ;(= aktuelle zeilen-
6c15 85 fa  -            sta  xsl     ;koordinaten plus
6c17 a5 8c  -            lda  xzh     ;abstand von einer
6c19 65 fd  -            adc  xabzh   ;zeile zur
6c1b 85 fb  -            sta  xsh     ;naechsten)
6c1d 60     -            rts
            -;
6c1e a5 fc  -nextspa     lda  ys      ;startkoordinaten
6c20 18     -            clc
6c21 65 b7  -            adc  yabs    ;der naechsten
6c23 85 fc  -            sta  ys
6c25 a5 fa  -            lda  xsl     ;spalte setzen
6c27 18     -            clc
6c28 65 b5  -            adc  xabsl   ;(=aktuelle spalten-
6c2a 85 fa  -            sta  xsl     ;koordinaten plus
6c2c a5 fb  -            lda  xsh     ;abstand von einer
6c2e 65 b6  -            adc  xabsh   ;spalte zur
6c30 85 fb  -            sta  xsh     ;naechsten)
6c32 60     -            rts
            -;
6c33 a5 8e  -nextzeichlda  y          ;startkoordinaten
6c35 18     -            clc
6c36 65 ba  -            adc  yab     ;des naechsten
6c38 85 8e  -            sta  y
6c3a a5 95  -            lda  xl      ;zeichens setzen
```

```
6c3c 18        -         clc
6c3d 65 b8     -         adc  xabl     ;(= linke, obere
6c3f 85 95     -         sta  xl       ;ecke des aktuellen
6c41 a5 96     -         lda  xh       ;zeichens plus
6c43 65 b9     -         adc  xabh     ;abstand von einem
6c45 85 96     -         sta  xh       ;zeichen zum
6c47 60        -         rts           ;naechsten)
               -;
               -;
               -;zwei-byte-wert holen
               -;====================
               -;
6c48 20 fd ae-holpar     jsr  komma
6c4b 20 9e ad-           jsr  frmevl   ;ausdruck holen
6c4e 4c bf b1-           jmp  integer  ;nach integer
               -;
               -;ein-byte-wert holen
               -;====================
               -;
6c51 20 fd ae-holbyte    jsr  komma
6c54 20 9e b7-           jmp  byte     ;byte holen
```

Da der Quelltext ausführlich kommentiert ist, möchte ich nur auf einen Punkt zu sprechen kommen: Es ist natürlich ungünstig, bei den Vergrößerungsfaktoren mit Vor- und Nachkommaanteilen rechnen zu müssen. Um den Weg ins Fließkommaformat nicht gehen zu müssen, habe ich mich eines Tricks bedient: Durch die Multiplikation des Nachkommaanteils mit 256 konnte der Faktor ins Format Lowbyte/Highbyte umgewandelt werden, was natürlich viel angenehmer ist. Wenn man nun z.B. beim Y-Faktor feststellen will, welche Zeile wie oft ausgegeben werden muß, braucht man nur den Vorkommaanteil um eins herunterzuzählen, bis er die Null erreicht. In jedem Durchgang wird die entsprechende Zeile einmal ausgegeben. Hat der Vorkommaanteil die Null erreicht, wird der Faktor im Format Lowbyte/Highbyte hinzuaddiert, die Zeile jedoch nicht ausgegeben. Dies soll nun an einem Beispiel verdeutlicht werden. Wir wollen ein Zeichen in Y-Richtung um den Faktor 1,8 vergrößern. Der Vorkommaanteil des Faktors beträgt 1, der Nachkommaanteil*256 daher 205.

Schritt	Operation	Vorkomma (V)	Nachkomma (N)	gesetzte Zeile
1	V+1,N+205	1	205	–
2	V–1	0	205	0
3	V+1,N+205	2	154	–
4	V–1	1	154	1
5	V–1	0	154	1
6	V+1,N+205	2	103	–

Fortsetzung

Schritt	Operation	Vorkomma (V)	Nachkomma (N)	gesetzte Zeile
7	V–1	1	103	2
8	V-1	0	103	2
9	V+1,N+205	2	52	–
10	V–1	1	52	3
11	V–1	0	52	3
12	V+1,N+205	2	1	–
13	V–1	1	1	4
14	V–1	0	1	4
15	V+1,N+205	1	206	–
16	V–1	0	206	5
17	V+1,N+205	2	155	–
18	V–1	1	155	6
19	V–1	0	155	6
20	V+1,N+205	2	104	–
21	V–1	1	104	7
22	V–1	0	104	7

Sie sehen, daß vier von fünf Zeilen doppelt gesetzt wurden, wie es dem Faktor 1,8 auch entsprechen sollte. Dieses Beispiel, das den Arbeitsgang des Programms Schritt für Schritt aufschlüsselt, zeigt aber auch, daß der Rechenaufwand mit dem Vergrößerungsfaktor sehr stark zunimmt. Zum Schluß will ich Ihnen den Aufruf für diesen Textbefehl nicht vorenthalten:

```
SYS 27214,X,Y,M,"Text",ZX,ZY,SX,SY,XFV,XFN,YFV,YFN,ABX,ABY
```

X: Startpunkt des Textes (X-Koordinate)
Y: Startpunkt des Textes (Y-Koordinate)
M: Zeichenmodus (s.o.)
»Text«: Auszugebender String. Dabei sind folgende Steuerzeichen erlaubt:
CTRL+A: Schaltet auf Kleinschrift/Großschriftmodus
CTRL+B: Schaltet auf Großschrift/Grafikmodus
CLR: Löscht Grafikbildschirm
HOME: Setzt Cursor auf linke, obere Ecke
RVS-ON: Schaltet Reversschrift ein
RVS-OFF: Schaltet Reversschrift aus

ZX: Zeilenvektor in x-Richtung (normal: 0)
ZY: Zeilenvektor in y-Richtung (normal: 1)
SX: Spaltenvektor in x-Richtung (normal: 1)
SY: Spaltenvektor in y-Richtung (normal: 0)
XFV: X-Faktor Vorkomma (normal: 1)
XFN: X-Faktor Nachkomma (normal: 0)

YFV: Y-Faktor Vorkomma (normal: 1)
YFN: Y-Faktor Nachkomma (normal: 0)
XAB: Abstand von einem Zeichen zum nächsten in x (normal: 8)
YAB: Abstand von einem Zeichen zum nächsten in y (normal: 0)

Zum Schluß dieses Kapitels möchte ich Sie noch auf ein Grafik-Demoprogramm auf der beiliegenden Diskette aufmerksam machen. Laden Sie es bitte mit

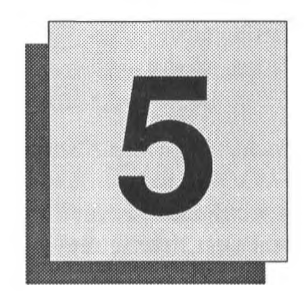
```
LOAD"GRAFIKDEMO",8
```

und starten es mit

```
RUN
```

In einem Menü können Sie sich dann wahlweise eine Demonstration der Befehle »Rechteck«, »Kreis/Ellipse« oder »Text« ansehen. Wahrscheinlich werden Sie erstaunt sein über die Geschwindigkeit der Zeichenbefehle sowie über die Einsatzmöglichkeiten des Text-Befehls. Sie sollten diese jedoch nur als Anregung für eigene Produktionen ansehen. So könnten Sie z.B. einen Linienalgorithmus entwerfen, der die Sonderfälle vertikale und horizontale Linie berücksichtigt. Hierzu könnten Sie die Rechteck-Routinen abwandeln. Sie können natürlich die Befehle auch in eigene, schon vorhandene Erweiterungen einbauen, um deren Qualität zu erhöhen.

Ich wünsche Ihnen jedenfalls viel Spaß bei der Grafikprogrammierung Ihres C64.



Programmierung von Basic-Erweiterungen

Daß der Befehlsumfang des Basic-Interpreters des C64 nicht gerade groß ist, ist nicht neu, aber unerfreulich. Daher machte man sich schon kurz nach Erscheinen des Computers an die Arbeit, ihn zu erweitern.

Für mich hat sich an dieser Stelle die Frage gestellt, ob es überhaupt sinnvoll ist, in einem Maschinensprachebuch über Basic-Erweiterungen zu sprechen, da diese ja im allgemeinen nur für Basic-Programmierer interessant sind. Zwei wichtige Gründe sprechen dafür: Zum einen gibt es viele hervorragende Programme, die ein Gemisch zwischen Basic und Maschinensprache darstellen, z.B. die erste Hi-Eddi-Version. Hierbei wird das Hauptprogramm in Basic ausgeführt und ruft dann die einzelnen Maschinenmodule auf. Ein Vorteil dieser Methode besteht u.a. darin, daß man das komplizierte Anlegen von Variablen in Basic erledigen kann, um dann die zeitkritischen Routinen, z.B. das Sortieren von Daten in Maschinensprache durchzuführen. Der zweite Grund besteht darin, daß der Basic-Interpreter selbst ja nichts anderes als ein Maschinenprogramm darstellt, was vom Basic aus ständig benutzt wird. In einer Modifikation des Basic-Interpreters ist also nichts anderes als eine Erweiterung eines internen Maschinenprogramms zu sehen, weshalb wir dieses Thema guten Gewissens behandeln können. Für uns bietet es sich natürlich an, die im letzten Kapitel besprochenen Grafikroutinen in das Basic einzubinden. Sie können die Routinen dann statt durch umständliche SYS-Befehle mit sinnvollen Namen aufrufen. Diese sollen lauten:

GRON – Einschalten der Grafik
GROFF – Ausschalten der Grafik
CLEAR – Löschen des Grafikschirms
COLOR – Setzen der Grafikfarben
PLOT – Zeichnen eines Punktes
ARC – Zeichnen eines Kreises/einer Ellipse
REC – Zeichnen eines Rechtecks
TEXT – Schreiben von Text in die Grafik

Der Basic-Interpreter weist für uns drei wichtige Vektoren auf, die die Einbindung der neuen Befehle erlauben. Da diese sich wie die schon kennengelernten Interrupt-Vektoren im RAM befinden, können wir sie abändern und auf unsere neuen Routinen zeigen lassen. Im einzelnen benötigen wir diese drei Vektoren:

$0304/$0305 – $A57C – Umwandlung in Interpretercode
$0306/$0307 – $A71A – Interpretercode in Klartext wandeln
$0308/$0309 – $A7E4 – Basic-Befehl ausführen

In den folgenden Abschnitten werden wir die Bedeutung und Aufgabe der einzelnen Vektoren kennenlernen. Zunächst wird jeweils die Original-ROM-Routine gelistet und erläutert. Anschließend können wir diese so ändern, daß unsere Grafikbefehle eingebunden werden. Auf der Diskette zum Buch ist der Quelltext komplett in einem Stück vorhanden. Wenn Sie ihn assembliert haben, müssen Sie ihn noch mit

```
SYS 27746
```

initialisieren. Die neuen Grafikbefehle stehen dann sofort zur Verfügung, wobei die neuen Auswerteroutinen direkt an die Grafikprogramme anschließen. Weiterhin befindet sich auch die komplette Befehlserweiterung auf der Diskette. Diese muß nur noch mit

```
LOAD "GRAFIK",8,1
```

geladen und mit SYS 27746 gestartet werden. Wie Sie die Erweiterung um zusätzliche Befehle ergänzen können, wird am Ende des Kapitels erläutert.

Im folgenden wird der zu dem jeweils zugehörigen Basic-Vektor gehörende Ausschnitt des Quelltextes erklärt. Auf eine ausführliche Besprechung der Initialisierungsroutine wollen wir verzichten, da diese in ihrer Funktion genau den Routinen entspricht, die den Interruptvektor geändert haben.

5.1 Die Umwandlung in Interpretercode

Eine wichtige Eigenschaft des Basic besteht darin, daß eine Programmzeile nicht so abgespeichert wird, wie sie eingegeben wurde. Jeder Basic-Befehl wird nämlich durch einen Ein-Byte-Wert, ein sogenanntes Token, abgekürzt. Damit wird zum einen eine gewaltige Speicherplatzersparnis erreicht, wie folgendes Beispiel zeigt: Der Ausdruck

```
PRINTPEEK(1)
```

würde normalerweise 12 Byte in Anspruch nehmen. Durch die Methode der Umwandlung in Token werden für die beiden Befehle PRINT und PEEK nur noch jeweils ein Byte benötigt, insgesamt also nur noch 5 Byte. Noch viel wichtiger ist aber die Tatsache, daß der Programmablauf deutlich beschleunigt wird. Wenn der Interpreter bei der Programmausführung auf ein Token stößt, kann er aus einer zugehörigen Tabelle sofort die Startadresse des entsprechenden Befehls ermitteln und diesen anspringen. Wäre der Befehl jedoch unabgekürzt gespeichert worden, müßte das Wort komplett mit einer Befehlstabelle verglichen werden. Nehmen Sie z.B. den Befehl RETURN. Hier müßten 6 (!) Zeichen abgearbeitet werden, man kann sich vorstellen, wie die ohnehin geringe Geschwindigkeit des Basic noch viel deutlicher reduziert

würde. Durch die Umwandlung in den Interpretercode schon bei der Eingabe einer Programmzeile ist dieser Vergleich nur noch einmal erforderlich und nicht bei jeder Programmausführung. Da die Eingabe von uns sowieso Schritt für Schritt durchgeführt wird, werden wir durch den Wandlungsvorgang kaum gestört.

Die Token der regulären Basic-Befehle belegen die Codes $80 bis einschließlich $CB. Am einfachsten wird die Sache für uns, wenn wir für unsere neuen Befehle die direkt anschließenden Token benutzen, also die Codes von $CC bis $FE. Die Zahl $FF dürfen wir nicht benutzen, da sie als Code der Zahl PI fungiert. Wir könnten so also 51 neue Befehle unterbringen. Sicherlich werden Sie fragen, wie es dann möglich ist, mehr als 100 neue Befehle zu integrieren. Hierbei wird die Sache dann etwas komplizierter. Die meisten Erweiterungen, wie z.B. Simons Basic, benutzen die 2-Byte-Token-Methode. Zunächst wird jedem Erweiterungsbefehl ein sogenanntes Erkennungstoken vorangestellt, das zwar nichts darüber aussagt, welcher Befehl angesprochen werden soll, jedoch den Befehl als Erweiterung kennzeichnet. Erst dann folgt das eigentliche Token für den Befehl. Damit kann man alle möglichen Codes von 0 bis 255 für seine Befehle benutzen. Eine weitere oft gestellte Frage ist die, warum beim C64 die Interpretercodes erst ab $80 beginnen. Nun, um ein Token von einem normalen Zeichen unterscheiden zu können, wird einfach das 7. Bit gesetzt. Anders ausgedrückt, jedes vom Interpreter angetroffene Zeichen, dessen 7. Bit gesetzt ist, muß ein Interpretercode sein. Das Erkennungstoken von Simons Basic genügt dieser Forderung ($CC). Die Auswerteroutinen von 2-Byte-Token-Erweiterungen sind jedoch wesentlich komplizierter als die von den normalen, da man hier die Original-ROM-Routinen nur minimal abzuändern braucht.

Bevor wir nun zu der Umwandlung in Interpretercode kommen, möchte ich zunächst beschreiben, was bei der Eingabe einer Zeile passiert.

Nach dem Einschalten befindet sich unser Rechner in der Eingabewarteschleife ab $A480. Die Eingabe einer Zeile wird als Unterprogramm ab Adresse $A560 ausgeführt. Hierbei werden alle eingegebenen Zeichen in dem sogenannten Eingabepuffer ab $0200 gespeichert. Dies geschieht so lange, bis entweder mehr als 88 Zeichen eingegeben wurden oder die RETURN-Taste gedrückt wurde. Danach wird der Puffer mit einem Null-Byte abgeschlossen, wodurch automatisch die Länge der Eingabe festgelegt ist. Nachdem diese beendet ist, wird in das Hauptprogramm, die Eingabewarteschleife, zurückgesprungen. Hier wird der Programmzeiger ($7A/$7B) auf ein Byte vor Beginn des Puffers ($01FF) gesetzt. Dieser wird in der CHRGET-Routine ab $0073, die nun angesprungen wird und ein Zeichen liest, um eins erhöht, so daß das erste Zeichen aus dem Eingabepuffer geholt wird. Ist dieses Null, ist der Puffer logischerweise leer und der bisher beschriebene Vorgang wird wiederholt.

Ist der Puffer nicht leer, wird als nächstes geprüft, ob eine Programmzeile oder ein Direktmodus-Befehl eingegeben wurde. Dazu wird das erste Zeichen auf Ziffer geprüft. Falls dies zutrifft, wird die komplette Nummer der Programmzeile geholt und eine eventuell schon existierende gelöscht. Nach dieser Aktion steht der Zeiger $7A/$7B auf dem ersten Zeichen hinter der Zeilennummer.

Jetzt sind wir endlich so weit, daß wir unsere Basic-Zeile umwandeln können. Dazu wird ein Unterprogramm ab $A579 angesprungen, dessen erster Befehl den indirekten Sprung

```
JMP ($0304)
```

darstellt. Dieser ist auf die Adresse direkt hinter ihm gerichtet ($A57C). Bevor wir nun durch Abänderung des Vektors dafür sorgen, daß auch unsere Grafikbefehle in Token umgewandelt werden, sehen wir uns die ROM-Routine einmal an:

```
, a57c   a6 7a      ldx $7a      ;Zeiger auf erstes Zeichen holen
, a57e   a0 04      ldy #$04     ;Zeiger in umgewandelte Zeile
, a580   84 0f      sty $0f      ;Flag für DATA-Modus loeschen
, a582   bd 00 02   lda $0200,x;Zeichen aus Puffer holen
, a585   10 07      bpl $a58e    ;Kein Basic-Code, auswerten
, a587   c9 ff      cmp #$ff     ;Code für PI ?
, a589   f0 3e      beq $a5c9    ;Ja, so uebernehmen
, a58b   e8         inx          ;Nein, Zeichen ignorieren
, a58c   d0 f4      bne $a582    ;Springt immer
, a58e   c9 20      cmp #$20     ;Leerzeichen ?
, a590   f0 37      beq $a5c9    ;Ja, so uebernehmen
, a592   85 08      sta $08      ;Zeichencode merken
, a594   c9 22      cmp #$22     ;Hochkomma ?
, a596   f0 56      beq $a5ee    ;Ja, so uebernehmen
, a598   24 0f      bit $0f      ;Flag gesetzt ?
, a59a   70 2d      bvs $a5c9    ;Ja, DATA-Modus, so uebernehmen
, a59c   c9 3f      cmp #$3f     ;Fragezeichen ?
, a59e   d0 04      bne $a5a4    ;Nein, weitermachen
, a5a0   a9 99      lda #$99     ;Ja, durch PRINT-Code ersetzen
, a5a2   d0 25      bne $a5c9    ;und so uebernehmen
, a5a4   c9 30      cmp #$30     ;Kleiner als "0" ?
, a5a6   90 04      bcc $a5ac    ;Ja, weitermachen
, a5a8   c9 3c      cmp #$3c     ;Kleiner als " " ?
, a5aa   90 1d      bcc $a5c9    ;Ja, so uebernehmen
, a5ac   84 71      sty $71      ;Zeiger in Zeile merken
, a5ae   a0 00      ldy #$00     ;Zaehler für Befehle auf Null
, a5b0   84 0b      sty $0b      ;setzen
, a5b2   88         dey
, a5b3   86 7a      stx $7a      ;Zeiger in Zeile merken
, a5b5   ca         dex
, a5b6   c8         iny          ;Zeiger für Befehle und Zeile
, a5b7   e8         inx          ;erhoehen
, a5b8   bd 00 02   lda $0200,x;Zeichen aus Puffer holen
, a5bb   38         sec          ;Durch Subtraktion mit Zeichen
, a5bc   f9 9e a0   sbc $a09e,y;für Befehlswort vergleichen
, a5bf   f0 f5      beq $a5b6    ;gleich, dann weitervergleichen
, a5c1   c9 80      cmp #$80     ;letzter Buchstabe (Bit 7 =1) ?
, a5c3   d0 30      bne $a5f5    ;Nein, naechstes Wort nehmen
, a5c5   05 0b      ora $0b      ;Ja, Token = $80+Befehlsnummer
```

```
, a5c7   a4 71      ldy $71      ;Zeiger in umgewandelte Zeile
, a5c9   e8         inx          ;Zeiger auf Original- und
, a5ca   c8         iny          ;umgewandelte Zeile erhoehen
, a5cb   99 fb 01   sta $01fb,y;Code abspeichern und wieder
, a5ce   b9 fb 01   lda $01fb,y;laden (Flags setzen)
, a5d1   f0 36      beq $a609    ;=0, dann Zeilenende
, a5d3   38         sec          ;Durch Subtraktion mit Doppel-
, a5d4   e9 3a      sbc #$3a     ;punkt vergleichen
, a5d6   f0 04      beq $a5dc    ;gleich, dann DATA-Modus loeschen
, a5d8   c9 49      cmp #$49     ;Code für DATA ?
, a5da   d0 02      bne $a5de    ;Wenn ja, DATA-Modus aktivieren
, a5dc   85 0f      sta $0f      ;(bei $49 ist Bit 6 gesetzt !)
, a5de   38         sec          ;Durch Subtraktion mit Code
, a5df   e9 55      sbc #$55     ;für REM vergleichen
, a5e1   d0 9f      bne $a582    ;Ungleich, naechstes Zeichen
, a5e3   85 08      sta $08      ;Nullbyte speichern
, a5e5   bd 00 02   lda $0200,x;naechstes Zeichen holen
, a5e8   f0 df      beq $a5c9    ;Null, dann Zeilenende
, a5ea   c5 08      cmp $08      ;Warten auf " oder Zeilenende
, a5ec   f0 db      beq $a5c9    ;Ja, Zeichen so uebernehmen
, a5ee   c8         iny          ;Weitere Zeichen solange
, a5ef   99 fb 01   sta $01fb,y;unveraendert uebernehmen, bis
, a5f2   e8         inx          ;entweder " oder Zeilenende
, a5f3   d0 f0      bne $a5e5    ;gefunden wird
, a5f5   a6 7a      ldx $7a      ;Zeilenzeiger auf Wortanfang
, a5f7   e6 0b      inc $0b      ;Zaehler auf naechstes Wort
, a5f9   c8         iny          ;Zeiger in Wort erhoehen
, a5fa   b9 9d a0   lda $a09d,y;Warten, bis Wort zuende ist
, a5fd   10 fa      bpl $a5f9    ;(d.h. Bit 7 ist gesetzt)
, a5ff   b9 9e a0   lda $a09e,y;Erstes Zeichen naechstes Wort
, a602   d0 b4      bne $a5b8    ;=0, dann Tabellenende
, a604   bd 00 02   lda $0200,x;in diesem Fall Zeichen
, a607   10 be      bpl $a5c7    ;unveraendert uebernehmen
, a609   99 fd 01   sta $01fd,y;Puffer mit 0 abschliessen
, a60c   c6 7b      dec $7b      ;Zeiger auf $01FF setzen
, a60e   a9 ff      lda #$ff
, a610   85 7a      sta $7a
, a612   60         rts
```

Diese recht kompliziert erscheinende Routine arbeitet folgendermaßen: Wie oben gesagt wurde, steht der Zeiger $7A/$7B auf dem ersten Zeichen hinter der Zeilennummer. Das X-Register hat in der gesamten Routine die Aufgabe, als Zeiger in der Original-Zeile zu dienen. Das Y-Register hingegen stellt den Zeiger in die umgewandelte Zeile dar. Diese verschiedenen Zeiger sind notwendig, da ja nach der Umwandlung pro Befehlswort nur noch ein Byte erforderlich ist. Damit wird die umgewandelte Zeile natürlich wesentlich kürzer als die Original-Zeile. Nun wird das Flag, das den DATA-Modus anzeigt, gelöscht. Jetzt wird das erste Zeichen aus dem Puffer geholt. Ist der Code größer als $7F, wird geprüft, ob es sich um den

Code für PI handelt, der unverändert übernommen wird. Alle anderen Zeichen, deren Code größer als $7F ist, werden ignoriert. Wenn Sie sich nämlich die Tabelle der ASCII-Codes ansehen, stellen Sie fest, daß ab $80 nur noch Zeichen folgen, deren Verwendung ausschließlich in Hochkommas sinnvoll ist. Dazu aber später.

Wenn nun der Code kleiner als $80 ist, beginnt eine Auswerteroutine. Diese prüft zunächst, ob es sich um ein Leerzeichen handelt, das unverändert übernommen wird. Sonst wird das Zeichen gespeichert. Der Grund hierfür liegt in folgendem: Im weiteren Verlauf wird nun geprüft, ob es sich um ein Hochkomma handelt. Trifft dies zu, müssen natürlich alle weiteren Zeichen unverändert übernommen werden, auch die mit einem Code größer als 127 (z.B. Grafikzeichen). Dies muß so lange geschehen, bis ein weiteres Hochkomma gefunden wird, das das Ende des Strings anzeigt. Hierfür wird jedes Zeichen einfach mit dem gespeicherten Code (hier: auch Hochkomma) verglichen. Auch beim Auftreten des Befehls DATA müssen alle weiteren Zeichen unverändert übernommen werden, diesmal bis ein Zeilenende oder ein Doppelpunkt erreicht wird. Der Doppelpunkt gibt ja an, daß ein neuer Befehl folgt. Dafür wird das Flag gesetzt, für den DATA-Modus ist Bit 6 relevant. Nun folgt die Umwandlung des Fragezeichens in den Code für den PRINT-Befehl. Weiterhin werden alle Zeichen, deren Code zwischen »0« und »< « liegt, unverändert übernommen. Dabei handelt es sich um die Ziffern sowie um die Zeichen »:« und »;«.

Hat das geprüfte Zeichen diese Prüfungen überstanden, wird nun auch der Teil eines Basic-Befehls geprüft. Dazu wird der Zeiger in die umgewandelte Zeile gerettet, da das Y-Register nun als Index für die Befehlstabelle der Basic-Befehle ab $A09E arbeitet. Der Vergleich wird so durchgeführt, daß vom Code des augenblicklichen Zeichens der des ersten Buchstabens des ersten Wortes abgezogen wird. Bei Gleichheit wird buchstabenweise weiterverglichen. Wird eine Differenz festgestellt, wird diese auf den Betrag $80 geprüft. Dies wäre nämlich das Zeichen dafür, daß das Ende des Wortes erreicht wurde. Der letzte Buchstabe jedes Wortes in der Tabelle wird nämlich mit gesetztem 7. Bit eingegeben, um das Wortende anzuzeigen. Durch den ODER-Befehl wird zu der $80 die aktuelle Befehlsnummer hinzuaddiert, die Summe stellt das Token dar, das nun gespeichert wird. Falls eine Ungleichheit festgestellt wird, muß zunächst das Wortende in der Tabelle abgewartet werden, bis dann mit dem nächsten Wort verglichen werden kann. Dies geschieht so lange, bis als erstes Zeichen eines neuen Wortes ein Nullbyte gefunden wird. Dies zeigt das Tabellenende an. Offenbar handelt es sich um keinen Basic-Befehl, weshalb das Zeichen unverändert übernommen wird.

An dieser Stelle möchte ich Ihnen das Phänomen mit den Abkürzungen erklären. Es erscheint z.B. merkwürdig, daß man den Befehl CLR mit C(Shift)L abkürzen kann, während für den Befehl CLOSE durch CL(Shift)O ein Buchstabe mehr erforderlich ist. Nun, das entscheidende ist, daß man ein Wort so abkürzen kann, daß es noch eindeutig identifizierbar ist. Zunächst muß man wissen, daß der Befehl CLR in der Tabelle vor dem CLOSE steht. Trifft nun unsere Routine auf die Abkürzung C(Shift)L, wird natürlich wie oben beschrieben die Tabelle durchsucht. Das erste Wort, das das Kriterium erfüllt, ein »C« am Anfang zu haben, ist der Befehl »CONT«. Nun wird das zweite Zeichen verglichen: Die Differenz von »O« und dem (Shift)L ist nicht gleich $80, weshalb »CONT« nicht in Frage kommt. Beim Vergleich mit dem Befehl

»CLR« jedoch tritt die Differenz -$80 = $80 auf. Es ist also egal, ob das geshiftete Zeichen in der Tabelle oder in unserem Wort auftaucht. Es wird daher angenommen, das »L« sei der letzte Buchstabe des Befehls, weshalb das Token für »CLR« übernommen wird. Wenn Sie nun den Befehl »CLOSE« mit C(Shift)L abkürzen würden, würde natürlich das erste Wort, das dieses Kriterium erfüllt, genommen, dies ist aber »CLR« ! Erst der dritte Buchstabe kann »CLOSE« eindeutig identifizieren. Theoretisch könnte man daher den einzigen Befehl, der mit einem »W« beginnt (WAIT), auch durch (Shift)W abkürzen. Dadurch würde jedoch die Prüfroutine gar nicht angesprungen, weil ja zunächst alle Zeichen mit gesetztem Bit 7 ignoriert werden!

Nach diesen sicherlich interessanten Zwischenbemerkungen nun wieder zu unserer ROM-Routine. Es beibt noch der Teil übrig, in dem die Zeichen so lange unverändert übernommen werden, bis ein bestimmtes Ereignis eintritt. Das Hochkomma und den DATA-Modus haben wir schon besprochen. Als letztes interessiert der REM-Befehl. Wenn dieser auftritt, wird ein Nullbyte als abzuwartendes Zeichen gesetzt. Dies bedeutet nichts anderes, als daß alle weiteren Zeichen bis zum Zeilenende so übernommen werden, während der DATA-Modus auch durch einen Doppelpunkt beendet werden kann.

Sind nun alle Zeichen des Eingabepuffers umgewandelt worden, wird dieser mit einem Nullbyte abgeschlossen. Der Zeiger $7A/$7B wird auf ein Zeichen vor dem Puffer gesetzt.

Mit diesen Informationen ist es natürlich nicht schwer, eine Erweiterungsroutine zu entwickeln. Wir übernehmen einfach die Original-Routine mit dem Unterschied, daß nach der (vergeblichen) Suche des Befehls in der Originaltabelle noch eine weitere durchsucht wird. Dies geschieht auf die gleiche Weise wie bei der ROM-Routine:

Listing: "basicerweiterung" (1. Teil)

```
6c81      -        .ba $6c81
          -;
          -;neue umwandlung in interpretercode
          -;==================================
          -;
0020      -        .eq leer    = $20 ;code fuer ' '
0022      -        .eq hoch    = $22 ;code fuer '"'
003f      -        .eq frage   = $3f ;code 'print'
0030      -        .eq null    = $30 ;code fuer '0'
003c      -        .eq kleiner = $3c ;code fuer ''
00ff      -        .eq pi      = $ff ;code fuer '='
003a      -        .eq doppel  = $3a ;code fuer ":"
0099      -        .eq print   = $99 ;token "print"
0083      -        .eq data    = $83 ;token "data"
008f      -        .eq rem     = $8f ;token "rem"
0049      -        .eq dado    = data-doppel
0055      -        .eq redo    = rem-doppel
000f      -        .eq flag    = $0f ;data-modus
000b      -        .eq zaehler = $0b ;zaehler befehle
```

```
6c81           -;
6c81 a6 7a     -          ldx  $7a       ;zeiger in puffer
6c83 a0 04     -          ldy  #04       ;zeiger in neue zeile
6c85 84 0f     -          sty  flag      ;data-modus loeschen
               -    ;                     (relevant bit 6 !)
6c87 bd 00 02-l1          lda  $0200,x ;zeichen holen
6c8a 10 07     -          bpl  aus       ;und auswerten
6c8c c9 ff     -          cmp  #pi       ;code fuer pi
6c8e f0 3e     -          beq  ueber     ;ja, so uebernehmen
6c90 e8        -          inx            ;zeichen ignorieren
6c91 d0 f4     -          bne  l1        ;sprint immer
               -;
6c93 c9 20     -aus       cmp  #leer     ;leerzeichen ?
6c95 f0 37     -          beq  ueber     ;ja, so uebernehmen
6c97 85 08     -          sta  $08       ;zeichencode merken
6c99 c9 22     -          cmp  #hoch     ;hochkomma ?
6c9b f0 56     -          beq  ueberl    ;ja, so uebernehmen
6c9d 24 0f     -          bit  flag      ;data-modus aktiv ?
6c9f 70 2d     -          bvs  ueber     ;ja, so uebernehmen
6ca1 c9 3f     -          cmp  #frage    ;fragezeichen?
6ca3 d0 04     -          bne  w1        ;nein, weiter
6ca5 a9 99     -          lda  #print    ;ja, durch code fuer
               -      ;                  ;print ersetzen
6ca7 d0 25     -          bne  ueber     ;springt immer
6ca9 c9 30     -w1        cmp  #null     ;kleiner als "0" ?
6cab 90 04     -          bcc  w2        ;ja, weiter
6cad c9 3c     -          cmp  #kleiner;kleiner als "" ?
6caf 90 1d     -          bcc  ueber     ;ja, so uebernehmen
6cb1 84 71     -w2        sty  $71       ;zeiger in umgewan-
6cb3 a0 00     -          ldy  #00       ;delte zeile merken
6cb5 84 0b     -          sty  zaehler ;zaehler fuer befehle
6cb7 88        -          dey            ;auf null setzen
6cb8 86 7a     -          stx  $7a       ;zeiger in zeile
6cba ca        -          dex            ;merken
6cbb c8        -l2        iny            ;x und y unveraendert
6cbc e8        -          inx            ;lassen (dey-iny=0)
6cbd bd 00 02-l3          lda  $0200,x ;zeichen holen
6cc0 38        -          sec            ;mit zeichen aus
6cc1 f9 9e a0-            sbc  $a09e,y ;befehl vergleichen
6cc4 f0 f5     -          beq  l2        ;=, weitervergleichen
6cc6 c9 80     -          cmp  #$80      ;letzter buchstabe
6cc8 d0 30     -          bne  next      ;nein, naechstes wort
6cca 05 0b     -          ora  zaehler ;befehlswort-nummer+
               -      ;                  ;$80 = token
6ccc a4 71     -w3        ldy  $71       ;zeiger zurueckholen
6cce e8        -ueber     inx
6ccf c8        -          iny            ;y mindestens 5
```

```
6cd0 99 fb 01-           sta  $01fb,y ;code abspeichern
6cd3 b9 fb 01-           lda  $01fb,y ;flags neu setzen
6cd6 f0 38    -          beq  end     ;0, dann zeilenende
6cd8 38       -          sec          ;zeichen mit doppel-
6cd9 e9 3a    -          sbc  #doppel ;punkt vergleichen
6cdb f0 04    -          beq  w4      ;data-modus loeschen
6cdd c9 49    -          cmp  #dado   ;gleich data-code"?"
6cdf d0 02    -          bne  w5      ;ja, data-modus
6ce1 85 0f    -w4        sta  flag    ;setzen
6ce3 38       -w5        sec
6ce4 e9 55    -          sbc  #redo   ;gleich rem-code ?
6ce6 d0 9f    -          bne  l1      ;nein, weiter
6ce8 85 08    -          sta  $08     ;ja, zeichen merken
6cea bd 00 02-l4         lda  $0200,x ;zeichen holen
6ced f0 df    -          beq  ueber   ;0, dann zeilenende
6cef c5 08    -          cmp  $08     ;warten auf '"'
6cf1 f0 db    -          beq  ueber   ;dann uebernehmen
6cf3 c8       -ueber1    iny
6cf4 99 fb 01-           sta  $01fb,y ;code speichern
6cf7 e8       -          inx
6cf8 d0 f0    -          bne  l4      ;springt immer
              -;
6cfa a6 7a    -next      ldx  $7a     ;zaehler auf
6cfc e6 0b    -          inc  zaehler ;naechsten befehl
6cfe c8       -l5        iny          ;warten, bis altes
6cff b9 9d a0-           lda  $a09d,y ;wort zuende ist
6d02 10 fa    -          bpl  l5
6d04 b9 9e a0-           lda  $a09e,y ;1.zeichen neues
6d07 d0 b4    -          bne  l3      ;wort, springt immer
6d09 f0 0f    -          beq  new     ;wenn tabelle noch
              -;                      ;nicht zuende ist
6d0b bd 00 02-no         lda  $0200,x ;zeichen holen
6d0e 10 bc    -          bpl  w3      ;und so uebernehmen
              -;
6d10 99 fd 01-end        sta  $01fd,y ;zeichen abspeichern
6d13 c6 7b    -          dec  $7b     ;zeiger auf
6d15 a9 ff    -          lda  #$ff    ;$01ff
6d17 85 7a    -          sta  $7a     ;(puffer -1)
6d19 60       -          rts
              -;
              -;verarbeitung neuer befehle
              -;=========================
              -;
6d1a a0 00    -new       ldy  #00     ;===============
6d1c b9 45 6d-           lda  tabnew,y
6d1f d0 02    -          bne  w6
6d21 c8       -l6        iny          ;funktion siehe
```

```
6d22 e8          -             inx
6d23 bd 00 02-w6               lda  $0200,x
6d26 38          -             sec
6d27 f9 45 6d-                 sbc  tabnew,y;programmteil
6d2a f0 f5       -             beq  16
6d2c c9 80       -             cmp  #$80
6d2e d0 04       -             bne  next1    ;w2 - w3
6d30 05 0b       -             ora  zaehler
6d32 d0 98       -             bne  w3       ;================
6d34 a6 7a       -next1        ldx  $7a
6d36 e6 0b       -             inc  zaehler
6d38 c8          -17           iny           ;funktion siehe
6d39 b9 44 6d-                 lda  tabnew-1,y
6d3c 10 fa       -             bpl  17       ;programmteil
6d3e b9 45 6d-                 lda  tabnew,y
6d41 d0 e0       -             bne  w6       ;next - exit
6d43 f0 c6       -             beq  no       ;=================
                 -;
                 -;tabelle der neuen befehle
                 -;========================
                 -;
6d45 47 52 4f-tabnew   .tx "groN"     ;grafik an
6d49 47 52 4f-         .tx "grofF"    ;grafik aus
6d4e 43 4f 4c-         .tx "coloR"    ;farbe setzen
6d53 43 4c 45-         .tx "cleaR"    ;grafik loeschen
6d58 50 4c 4f-         .tx "ploT"     ;punkt
6d5c 41 52 c3-         .tx "arC"      ;kreis/ellipse
6d5f 52 45 c3-         .tx "reC"      ;rechteck
6d62 54 45 58-         .tx "texT"     ;text
6d66 00          -     .by 0          ;Tabellenende
```

Da unsere Erweiterungstoken direkt an die normalen anschließen, müssen wir nicht einmal den Zähler für die Befehlsworte ändern. Sie stellen eine 100%ige Übereinstimmung zwischen der Funktion des Original- und des Erweiterungsteils fest.

Der Vorteil dieser Routine besteht darin, daß man sie noch erheblich erweitern kann. Dazu schreibt man einfach die weiteren erwünschten Befehle (natürlich alle mit gesetztem Bit 7 beim letzten Zeichen !) in die Tabelle und verschiebt das Nullbyte an ihr Ende. Insgesamt können 51 neue Befehle verarbeitet werden. Man muß jedoch auf eine Sache aufpassen: Da die Tabelle mit einem 8-Bit-Register indiziert wird, darf die Gesamtlänge aller Befehle nicht größer als 255 sein, d.h. pro Befehl fünf Zeichen. Natürlich kann man auch längere Wörter verwenden, muß dafür dann aber andere kürzen.

5.2 Die Umwandlung des Interpretercodes in Klartext

Wenn Sie einen neuen Befehl in ein Token umgewandelt haben, wie es im letzten Abschnitt beschrieben wurde, stehen Sie vor dem Problem, daß Sie nach dem LIST-Befehl statt dem Befehlswort nur ein undefiniertes Zeichen auf dem Bildschirm sehen. Dieser kann natürlich nicht »wissen«, daß neue Befehle hinzugekommen sind. Deshalb müssen wir offenbar auch hier eine Erweiterung schreiben.

Die eigentliche LIST-Routine steht von $A69C bis $A741 im Basic-ROM. Dabei hat der Teil von $A69C bis $$A716 die Aufgabe, z.B. die Parameter hinter dem Befehlswort LIST auszuwerten, die Adressen der Zeilennummern zu holen und diese auszugeben. Alles in allem wäre es unsinnig, wenn man auch diesen Teil der Routine neu schreiben müßte, da sich an ihm überhaupt nichts ändern würde. Vielmehr ist nur notwendig, den Teil ab $A717 zu modifizieren, da in ihm die Umwandlung der Token in Klartext durchgeführt wird. Zum Glück haben das auch die Entwickler des Basic-Interpreters erkannt und den Sprungvektor

```
JMP ($0306)
```

erst an dieser Stelle eingebaut. Er zeigt normalerweise direkt an die folgende Adresse $A71A. Beim Aufruf der Routine steht das zu listende Zeichen im Akku. Nun wollen wir uns die Funktionsweise der ROM-Routine einmal ansehen:

```
, a71a   10 d7      bpl $a6f3  ;kein Basic-Code, so ausgeben
, a71c   c9 ff      cmp #$ff   ;Code für PI ?
, a71e   f0 d3      beq $a6f3  ;Ja, so ausgeben
, a720   24 0f      bit $0f    ;Hochkommamodus aktiv ?
, a722   30 cf      bmi $a6f3  ;Ja, Zeichen so ausgeben
, a724   38         sec        ;Offset für Token abziehen
, a725   e9 7f      sbc #$7f   ;Nummer des Befehlswortes
, a727   aa         tax        ;als Zaehler nach X
, a728   84 49      sty $49    ;Y-Register retten
, a72a   a0 ff      ldy #$ff   ;Y auf Null setzen ($FF+1)
, a72c   ca         dex        ;Befehlswort gefunden ?
, a72d   f0 08      beq $a737  ;Ja, zur Zeichenausgebe
, a72f   c8         iny        ;Zaehler fuer Zeichen erhoehen
, a730   b9 9e a0   lda $a09e,y;Zeichen holen
, a733   10 fa      bpl $a72f  ;letzter Buchstabe ?
, a735   30 f5      bmi $a72c  ;Ja, naechstes Wort
, a737   c8         iny        ;naechstes Zeichen holen
, a738   b9 9e a0   lda $a09e,y;wenn Wort zuende, dann
, a73b   30 b2      bmi $a6ef  ;fertig
, a73d   20 47 ab   jsr $ab47  ;Zeichen ausgeben
, a740   d0 f5      bne $a737  ;unbedingter Sprung
```

Zunächst wird an Hand des Bit 7 geprüft, ob das Zeichen überhaupt ein Interpretercode ist. Falls dies nicht der Fall ist oder es sich um den Code für PI handelt, wird es unverändert ausgegeben. Im letzten Abschnitt wurde durch das Bit 6 der Speicherzelle $0F der DATA-Modus angezeigt. Hier erkennt man, daß Bit 7 analog den Hochkommamodus signalisiert. Falls er aktiv ist, wird das Zeichen natürlich auch unverändert ausgegeben, da es sich um einen Teil eines Strings handelt.

Nun kommt der Teil, der bei »echten« Token durchlaufen wird. Zunächst wird vom Interpretercode der Offset $7F abgezogen. Dadurch werden die Token in den Bereich von 1–76 verschoben. Der so neu entstandene Code wird als Zähler benutzt: Die Befehlstabelle wird nun Wort für Wort durchsucht. Jedesmal, wenn ein Wortende gefunden wird, wird der Zähler um eins heruntergezählt. Offenbar steht der Zeiger dann auf dem ersten Zeichen des gesuchten Wortes, wenn er auf Null heruntergezählt wurde. Dann können die Zeichen so lange ausgegeben werden, bis wiederum auf das Wortende gestoßen wird, das durch gesetztes Bit 7 erkennbar ist. Jetzt ist die Umwandlung beendet und es wird in die LIST-Routine zurückgesprungen.

Um unsere neuen Befehlswörter zu listen, können wir genauso vorgehen. Wir brauchen nur zu prüfen, ob das Token größer als $CB ist. Falls das zutrifft, müssen wir nur den Offset auf $CB abändern und können dann genau das Verfahren der ROM-Routine anwenden:

Listing: "basicerweiterung" (2. Teil)

```
6d67          -         .ba $6d67
              -;list-routine fuer neue befehle
              -;==============================
              -;
ab47          -         .eq byout   = $ab47 ;zeichenausgabe
a724          -         .eq oldtok  = $a724 ;list der nor-
              -;                            ;malen befehle
a6f3          -         .eq bylist  = $a6f3 ;byte ausgeben
a6ef          -         .eq oldlist= $a6ef  ;alte routine
              -;
6d67 10 0f    -         bpl  normal  ;kein token
6d69 24 0f    -         bit  flag    ;hochkommamodus ?
6d6b 30 0b    -         bmi  normal  ;ja, normal ausgeben
6d6d c9 ff    -         cmp  #pi     ;code fuer pi
6d6f f0 07    -         beq  normal  ;ja, normal ausgeben
6d71 c9 cc    -         cmp  #$cc    ;neuer befehl ?
6d73 b0 06    -         bcs  toknew  ;ja, listen
6d75 4c 24 a7-          jmp  oldtok  ;alten befehl listen
6d78 4c f3 a6-normal    jmp  bylist  ;byte ausgeben
              -;
6d7b 38       -toknew   sec
6d7c e9 cb    -         sbc  #$cb    ;offset abziehen
6d7e aa       -         tax          ;ergebnis = zaehler
6d7f 84 49    -         sty  $49     ;y retten
```

```
6d81 a0 ff    -          ldy  #$ff      ;wegen "iny" = 0
6d83 ca          -w7     dex            ;herunterzaehlen
6d84 f0 08    -          beq  ok        ;=0, dann gefunden
6d86 c8          -18     iny            ;warten, bis wort
6d87 b9 45 6d-           lda  tabnew,y;zuende ist
6d8a 10 fa    -          bpl  18        ;(bit 7 gesetzt)
6d8c 30 f5    -          bmi  w7        ;naechstes wort
                 -;
6d8e c8          -ok     iny            ;wort gefunden
6d8f b9 45 6d-           lda  tabnew,y;zeichen holen
6d92 30 05    -          bmi  ende      ;letztes zeichen
6d94 20 47 ab-           jsr  byout     ;ausgeben
6d97 d0 f5    -          bne  ok        ;springt immer
                 -;
6d99 4c ef a6-ende       jmp  oldlist ;zur List-Routine
```

Die Tabelle »Tabnew« ist natürlich mit der Befehlstabelle aus dem letzten Abschnitt identisch. Da in der Praxis ja beide Programmteile zusammen assembliert werden, wird eine eventuelle Erweiterung dieser Tabelle natürlich auch automatisch von der List-Routine mitverarbeitet. Sie können den obigen Teil daher auf jeden Fall unverändert verwenden, egal wieviele neue Befehle Sie einsetzen möchten.

5.3 Die Ausführung der Basic-Befehle

Bislang haben wir nur erreicht, daß die neuen Befehle in Interpretercode umgewandelt und beim Listen wieder als Klartext dargestellt werden. Jeder Versuch, einen neuen Befehl auszuführen, würde jedoch mit einem »SYNTAX ERROR« des Interpreters quittiert. Wir müssen zunächst noch die Routine zur Befehlsausführung ändern. Dazu ist in der Interpreterschleife von $A7AE bis $A7EC ein weiterer indirekter Sprungbefehl an der Adresse $A7E1 integriert:

```
JMP ($0308)
```

Er zeigt wie üblich direkt auf die folgende Adresse ($A7E4). Zunächst wird ein Zeichen mit der CHRGET-Routine geholt. Dann wird die Routine zur Befehlsausführung ($A7ED-$A81C) als Unterprogramm angesprungen, um dann wieder an den Anfang der Interpreterschleife zurückzukehren. Für uns interessant ist natürlich nur der Teil, der für die Befehlsausführung zuständig ist:

```
, a7ed   f0 3c      beq $a82b  ;Zeilenende, dann fertig
, a7ef   e9 80      sbc #$80   ;Token ?
, a7f1   90 11      bcc $a804  ;Nein, zum LET-Befehl
, a7f3   c9 23      cmp #$23   ;Funktions-Token oder GOTO-
, a7f5   b0 17      bcs $a80e  ;Befehl
, a7f7   0a         asl  a     ;Basic-Code mal 2
```

```
, a7f8   a8         tay         ;als Index fuer Tabelle
, a7f9   b9 0d a0   lda $a00d,y;Befehlsadresse - 1 holen
, a7fc   48         pha         ;und auf Stack schieben
, a7fd   b9 0c a0   lda $a00c,y;(erst High-Byte, dann
, a800   48         pha         ;Low-Byte)
, a801   4c 73 00   jmp $0073   ;naechstes Zeichen und ausfuehren
, a804   4c a5 a9   jmp $a9a5   ;zum LET-Befehl
```

Zunächst wird festgestellt, ob es sich um ein Null-Byte und damit um das Zeilenende handelt. Ist dies nicht der Fall, wird ein Offset von $80 abgezogen. Die Token sind dadurch in den Bereich von 0 bis 75 verschoben worden. Jetzt wird geprüft, ob der Differenzcode größer als $22 ist, d.h. ob der Originalcode größer als $A2 ist. Zunächst kann man dadurch die Token der Basic-Funktionen von denen der Befehle unterscheiden: Während die Befehle die Codes von $80–$A9 belegen, nehmen die Funktionen die von $B4 bis $CA ein. Falls es sich also um ein Funktions-Token handelt, wird eine andere Routine angesprungen. Es gibt jedoch noch einen anderen Sonderfall: Dies ist der Basic-Befehl »GOTO«, wenn er nicht zusammenhängend, sondern durch ein Leerzeichen getrennt eingegeben wird: »GO TO«. In diesem Fall wird bei der Umwandlung in Interpretercode nicht das Token für den Befehl GOTO ($89) erzeugt, sondern zwei Codes, da sowohl der Teil »GO« als auch der Teil »TO« jeweils ein eigenes Token aufweisen: Während das »TO« (Code: $A4) auch noch in FOR-TO-Schleifen verwendet wird, dient das »GO« ausschließlich dazu, bei getrennter Eingabe von GOTO keinen Fehler zu erzeugen. Wenn nun ein Token eines »richtigen« Basic-Befehls gefunden wurde, werden aus einer Tabelle jeweils High- und Lowbyte minus eins der Startadresse des Befehls ausgelesen und auf den Stack geschoben. Dann wird die CHRGET-Routine mit dem JMP-Befehl angesprungen. Durch die Stackoperation wird nun nach Beendigung der CHRGET-Routine mit dem Befehl RTS aus einem vermeintlichen Unterprogramm zurückgesprungen, dessen Rücksprungadresse die Startadresse des Befehls darstellt. Da diese durch den Befehl RTS automatisch um eins erhöht wird, muß in der Tabelle unbedingt die Adresse um eins vermindert angegeben werden. Die eigentliche Befehls-Routine muß ebenfalls mit einem RTS abgeschlossen werden, damit aus dem Unterprogramm ab $A7ED in die Interpreterschleife zurückgesprungen werden kann.

In unserer neuen Routine werden wir wie bislang auch dem Verfahren des Interpreters folgen. Dazu prüfen wir zunächst, ob das Token im Bereich von $CC bis $FE liegt und verzweigen dann entsprechend in eine Routine, welche die Startadresse des neuen Befehls auf den Stack schiebt und den Befehl ausführt:

Listing: "basicerweiterung" (3. Teil)

```
6d9c     -          .ba $6d9c
         -;
         -;ausfuehrung der neuen befehle
         -;=============================
         -;
a7ed     -          .eq befold = $a7ed ;ausfuehrung
```

```
00cc           -         .eq firstb = $cc   ;1. fremdtoken
00fe           -         .eq lastb  = $fe   ;letztes
a7ae           -         .eq loop   = $a7ae ;interpreter
0073           -         .eq chrget = $0073 ;holt zeichen
0079           -         .eq chrgot = $0079 ;aus basic-text
               -;
6d9c 20 73 00-           jsr  chrget  ;token holen
6d9f 20 a5 6d-           jsr  do      ;befehl ausfuehren
6da2 4c ae a7-           jmp  loop    ;und beenden
               -;
6da5 c9 cc     -do       cmp  #firstb ;kleiner als $cc
6da7 90 13     -         bcc  tokold  ;ja, altes token
6da9 c9 ff     -         cmp  #lastb+1;groesser als $fe
6dab b0 0f     -         bcs  tokold  ;ja, altes token
               -;
               -;neue befehlsausfuehrung
               -;=====================
               -;
6dad 38        -         sec          ;(token - startwert)
6dae e9 cc     -         sbc  #firstb ;= zeiger in tabelle
6db0 aa        -         tax
6db1 bd c2 6d-           lda  hightab,x;ruecksprungadresse
6db4 48        -         pha          ;high-byte und
6db5 bd ca 6d-           lda  lowtab,x;low-byte auf stack
6db8 48        -         pha
6db9 4c 73 00-           jmp  chrget  ;zeichen holen
               -;
6dbc 20 79 00-tokold     jsr  chrgot  ;alten befehl
6dbf 4c ed a7-           jmp  befold  ;ausfuehren
               -;
               -;Tabelle mit Startadressen der Befehle
               -;===================================
               -;
6dc2 5f        -hightab  .by ($6000-1)        ;gron
6dc3 60        -         .by ($6018-1)        ;groff
6dc4 60        -         .by ($6030+3-1)      ;color
6dc5 60        -         .by ($606e-1)        ;clear
6dc6 63        -         .by ($6377+3-1)      ;plot
6dc7 63        -         .by ($638b+3-1)      ;arc
6dc8 68        -         .by ($685d+3-1)      ;rec
6dc9 6a        -         .by ($6a4e+3-1)      ;text
               -;
6dca ff        -lowtab   .by ($6000-1)        ;gron
6dcb 17        -         .by ($6018-1)        ;groff
6dcc 32        -         .by ($6030+3-1)      ;color
6dcd 6d        -         .by ($606e-1)        ;clear
6dce 79        -         .by ($6377+3-1)      ;plot
```

```
270:    6dcf 8d       -         .by ($638b+3-1)         ;arc
271:    6dd0 5f       -         .by ($685d+3-1)         ;rec
272:    6dd1 50       -         .by ($6a4e+3-1)         ;text
273:                  -;
```

Sie erkennen, daß die neue Routine doch einen kleinen Unterschied zur ROM-Routine enthält: Während bei dieser die High- und Lowbytes in einer Tabelle jeweils direkt aufeinanderfolgten, wurden hier zwei separate Tabellen für die jeweils höher- und niederwertigen Adreßbytes angelegt. Wie man hier vorgeht, ist Geschmackssache, das Ergebnis ist immer das gleiche.

In den Tabellen selbst erkennt man die Startadressen der SYS-Befehle aus dem letzten Kapitel »Grafik« wieder. Sie wurden, wie oben begründet, jeweils um 1 Byte reduziert. Zu einigen Adressen wurden jedoch drei weitere Byte hinzuaddiert. Dies ist immer dann erforderlich, wenn nach dem SYS-Befehl ein Komma folgte, um die Parameter von der Adresse abzutrennen. Die Routine, die ein Zeichen aus dem Basic-Text holt und dies auf Komma prüft, wird mit dem Befehl

```
JSR $AEFD
```

aufgerufen. Ein Komma als Trennzeichen ist immer zwischen zwei Basic-Ausdrücken erforderlich, nicht jedoch zwischen einem Basic-Befehl und einem Ausdruck. Der Nachteil des SYS-Befehls besteht darin, daß nach ihm ja schon ein Ausdruck ausgewertet werden muß, nämlich die Startadresse des Maschinenprogramms. Um diese von den übrigen Parametern abzutrennen, muß ein Komma eingefügt werden. Wenn wir nun den Befehl über ein Token aufrufen, kann das Komma natürlich entfallen. Damit ist auch der Aufruf des Kommaprüf-Unterprogramms überflüssig geworden. Um ihn zu überspringen, wurde die Startadresse des Befehls einfach um drei heraufgesetzt. Jetzt wissen Sie auch, warum ich bei den Grafikroutinen stets als ersten Befehl die Kommaprüfung ausgeführt habe, obwohl die Routinen, die die Parameter aus dem Basic-Text holen, oft erst am Ende des Befehls stehen. So mußten die einzelnen Grafikroutinen überhaupt nicht geändert werden:

GRON	– schaltet Grafik ein
GROFF	– schaltet Textmodus ein
COLOR PF,HF	– setzt Farbe des Grafikschirms
CLEAR	– löscht Grafikschirm
PLOT X,Y,M	– zeichnet Punkt
ARC XM,YM,M,XR,YR,AW,EW	– zeichnet Kreis/Ellipse
REC X,Y,M,DX,DY	– zeichnet Rechteck
TEXT X,Y,M,A$,ZX,ZY,SX,SY,XFV, XFN,YFV,YFN,ABX,ABY	– schreiben

Anhang 1

Umrechnungstabelle Dezimal – Hexadezimal – Binär

dez	hex	binär	dez	hex	binär	dez	hex	binär
0	00	00000000	26	1A	00011010	52	34	00110100
1	01	00000001	27	1B	00011011	53	35	00110101
2	02	00000010	28	1C	00011100	54	36	00110110
3	03	00000011	29	1D	00011101	55	37	00110111
4	04	00000100	30	1E	00011110	56	38	00111000
5	05	00000101	31	1F	00011111	57	39	00111001
6	06	00000110	32	20	00100000	58	3A	00111010
7	07	00000111	33	21	00100001	59	3B	00111011
8	08	00001000	34	22	00100010	60	3C	00111100
9	09	00001001	35	23	00100011	61	3D	00111101
10	0A	00001010	36	24	00100100	62	3E	00111110
11	0B	00001011	37	25	00100101	63	3F	00111111
12	0C	00001100	38	26	00100110	64	40	01000000
13	0D	00001101	39	27	00100111	65	41	01000001
14	0E	00001110	40	28	00101000	66	42	01000010
15	0F	00001111	41	29	00101001	67	43	01000011
16	10	00010000	42	2A	00101010	68	44	01000100
17	11	00010001	43	2B	00101011	69	45	01000101
18	12	00010010	44	2C	00101100	70	46	01000110
19	13	00010011	45	2D	00101101	71	47	01000111
20	14	00010100	46	2E	00101110	72	48	01001000
21	15	00010101	47	2F	00101111	73	49	01001001
22	16	00010110	48	30	00110000	74	4A	01001010
23	17	00010111	49	31	00110001	75	4B	01001011
24	18	00011000	50	32	00110010	76	4C	01001100
25	19	00011001	51	33	00110011	77	4D	01001101

dez	hex	binär
78	4E	01001110
79	4F	01001111
80	50	01010000
81	51	01010001
82	52	01010010
83	53	01010011
84	54	01010100
85	55	01010101
86	56	01010110
87	57	01010111
88	58	01011000
89	59	01011001
90	5A	01011010
91	5B	01011011
92	5C	01011100
93	5D	01011101
94	5E	01011110
95	5F	01011111
96	60	01100000
97	61	01100001
98	62	01100010
99	63	01100011
100	64	01100100
101	65	01100101
102	66	01100110
103	67	01100111
104	68	01101000
105	69	01101001
106	6A	01101010
107	6B	01101011
108	6C	01101100
109	6D	01101101
110	6E	01101110
111	6F	01101111
112	70	01110000
113	71	01110001
114	72	01110010
115	73	01110011
116	74	01110100
117	75	01110101
118	76	01110110

dez	hex	binär
119	77	01110111
120	78	01111000
121	79	01111001
122	7A	01111010
123	7B	01111011
124	7C	01111100
125	7D	01111101
126	7E	01111110
127	7F	01111111
128	80	10000000
129	81	10000001
130	82	10000010
131	83	10000011
132	84	10000100
133	85	10000101
134	86	10000110
135	87	10000111
136	88	10001000
137	89	10001001
138	8A	10001010
139	8B	10001011
140	8C	10001100
141	8D	10001101
142	8E	10001110
143	8F	10001111
144	90	10010000
145	91	10010001
146	92	10010010
147	93	10010011
148	94	10010100
149	95	10010101
150	96	10010110
151	97	10010111
152	98	10011000
153	99	10011001
154	9A	10011010
155	9B	10011011
156	9C	10011100
157	9D	10011101
158	9E	10011110
159	9F	10011111

dez	hex	binär
160	A0	10100000
161	A1	10100001
162	A2	10100010
163	A3	10100011
164	A4	10100100
165	A5	10100101
166	A6	10100110
167	A7	10100111
168	A8	10101000
169	A9	10101001
170	AA	10101010
171	AB	10101011
172	AC	10101100
173	AD	10101101
174	AE	10101110
175	AF	10101111
176	B0	10110000
177	B1	10110001
178	B2	10110010
179	B3	10110011
180	B4	10110100
181	B5	10110101
182	B6	10110110
183	B7	10110111
184	B8	10111000
185	B9	10111001
186	BA	10111010
187	BB	10111011
188	BC	10111100
189	BD	10111101
190	BE	10111110
191	BF	10111111
192	C0	11000000
193	C1	11000001
194	C2	11000010
195	C3	11000011
196	C4	11000100
197	C5	11000101
198	C6	11000110
199	C7	11000111
200	C8	11001000

dez	hex	binär
201	C9	11001001
202	CA	11001010
203	CB	11001011
204	CC	11001100
205	CD	11001101
206	CE	11001110
207	CF	11001111
208	D0	11010000
209	D1	11010001
210	D2	11010010
211	D3	11010011
212	D4	11010100
213	D5	11010101
214	D6	11010110
215	D7	11010111
216	D8	11011000
217	D9	11011001
218	DA	11011010
219	DB	11011011

dez	hex	binär
220	DC	11011100
221	DD	11011101
222	DE	11011110
223	DF	11011111
224	E0	11100000
225	E1	11100001
226	E2	11100010
227	E3	11100011
228	E4	11100100
229	E5	11100101
230	E6	11100110
231	E7	11100111
232	E8	11101000
233	E9	11101001
234	EA	11101010
235	EB	11101011
236	EC	11101100
237	ED	11101101
238	EE	11101110

dez	hex	binär
239	EF	11101111
240	F0	11110000
241	F1	11110001
242	F2	11110010
243	F3	11110011
244	F4	11110100
245	F5	11110101
246	F6	11110110
247	F7	11110111
248	F8	11111000
249	F9	11111001
250	FA	11111010
251	FB	11111011
252	FC	11111100
253	FD	11111101
254	FE	11111110
255	FF	11111111

Anhang 2

Alphabetische Tabelle der Prozessorbefehle und Opcodes

Mnem.	–	zp	zx	zy	ab	ax	ay	ix	iy	im
a11						9c	9e			
aax		87		97	8f					
adc		65	75		6d	7d	79	61	71	69
and		29	35		2d	3d	39	21	31	29
asl	0a	06	16		0e	1e				
asr										6b
arr										4b
axs										cb
bcc	90									
bcs	b0									
beq	f0									
bit		24			2c					
bmi	30									
bne	d0									
bpl	10									
brk	00									
bvc	50									
bvs	70									
clc	18									
cld	d8									

Mnem.	**–**	**zp**	**zx**	**zy**	**ab**	**ax**	**ay**	**ix**	**iy**	**im**
cli	58									
clv	b8									
cmp		c5	d5		cd	dd	d9	c1	d1	c9
cpx		e4			ec					e0
cpy		c4			cc					c0
dcp		c7	d7		cf	df	db	c3	d3	
dec	c6	d6	ce	de						
dex	ca									
dey	88									
dop	*)									
eor		45	55		4d	5d	59	41	51	49
inc		e6	f6		ee	fe				
inx	8									
iny	c8									
isc		e7	f7		ef	ff	fb	e3	f3	
jmp					4c			6c		
jsr					20					
kil	*)									
lar				bb						
lax –		a7	b7		af		bf	a3	b3	
lda	a5	b5		ad	bd	b9	a1	b1	a9	
ldx		a6		b6	ae		be			a2
ldy		a4	b4		ac	bc				a0
lsr	4a	46	56		4e	5e				
nop	ea									
nop	*)									
ora		05	15		0d	1d	19	01	11	09
pha	48									
php	08									
pla	68									
plp	28									
rla		27	37		2f	3f	3b	23	33	

Mnem.	**–**	**zp**	**zx**	**zy**	**ab**	**ax**	**ay**	**ix**	**iy**	**im**
rol	2a	26	36		2e	3e				
ror	6a	66	76		6e	7e				
rra		67	77		6f	7f	7b	63	73	
rti	40									
rts	60									
sbc		e5	f5		ed	fd	f9	e1	f1	f9
sec	38									
sed	f8									
sei	78									
slo		07	17		0f	1f	1b	13	03	
sre		47	57		4f	5f	5b	43	53	
sta		85	95		8d	9d	99	81	91	
stx		86		96	8e					
sty		84	94		8c					
tax	aa									
tay	a8									
top	*)									
tsx	ba									
txa	8a									
txs	9a									
tya	98									

Die gerasterten Opcodes sind illegal.

Die mit *) gekennzeichneten Befehle haben mehrere gleichwertige Opcodes:
dop: 04, 14, 34, 44, 54, 64, 74, d4, f4, 80, 89, 93
kil: 02, 12, 22, 32, 42, 52, 62, 72, 92, b2, d2, f2
nop: 1a, 3a, 5a, 7a, da, fa
top: 0c, 1c, 3c, 5c, 7c, dc, fc

Adressierungsarten:

–	=	implizt oder Akkumulator, bei Branches entsprechend eine Adreßdistanz
zp	=	Zeropage absolut
zx	=	Zeropage X-indiziert
zy	=	Zeropage Y-indiziert
ab	=	absolut
ax	=	absolut X-indiziert
ay	=	absolut Y-indiziert
ix	=	indiziert indirekt; bei Jump indir. Sprung
iy	=	indirekt indiziert
im	=	unmittelbar

Anhang 3

Nach Wert sortierte Übersicht über die Prozessorbefehle inklusive illegaler Opcodes

Opcode hex	Opcode dez	Befehl
00	0	brk
01	1	ora ix
02	2	kil
03	3	slo iy
04	4	dop
05	5	ora zp
06	6	asl zp
07	7	slo zp
08	8	php
09	9	ora im
0A	10	asl a
0B	11	top
0C	12	–
0D	13	ora ab
0E	14	asl ab
0F	15	slo ab
10	16	bpl rel
11	17	ora iy
12	18	kil
13	19	slo ix
14	20	dop
15	21	ora zx
16	22	asl zx
17	23	slo zx
18	24	clc
19	25	ora ay
1A	26	nop
1B	27	slo ay
1C	28	top
1D	29	ora ax
1E	30	asl ax
1F	31	slo ax
20	32	jsr ab
21	33	and ix
22	34	kil
23	35	rla ix
24	36	bit zp
25	37	and zp
26	38	rol zp
27	39	rla zp
28	40	plp
29	41	and im
2A	42	rol a
2B	43	–
2C	44	bit ab
2D	45	and ab
2E	46	rol ab
2F	47	rla ab
30	48	bmi rel
31	49	and ix
32	50	kil
33	51	rla iy
34	52	dop
35	53	and zx
36	54	rol zx
37	55	rla zx
38	56	sec
39	57	and ay
3A	58	nop
3B	59	rla ay
3C	60	top
3D	61	and ax
3E	62	rol ax
3F	63	rla ax
40	64	rti
41	65	eor ix
42	66	kil
43	67	sre ix
44	68	dop
45	69	eor zp
46	70	lsr zp
47	71	sre zp
48	72	pha
49	73	eor im
4A	74	lsr a

Opcode hex	dez	Befehl
4B	75	arr im
4C	76	jmp ab
4D	77	eor ab
4E	78	lsr ab
4F	79	sre ab
50	80	bvc rel
51	81	eor iy
52	82	kil
53	83	sre iy
54	84	dop
55	85	eor zx
56	86	lsr zx
57	87	sre zx
58	88	cli
59	89	eor ay
5A	90	nop
5B	91	sre ay
5C	92	top
5D	93	eor ax
5E	94	lsr ax
5F	95	sre ax
60	96	rts
61	97	adc ix
62	98	kil
63	99	rra ix
64	100	dop
65	101	adc zp
66	102	ror zp
67	103	rra zp
68	104	pla
69	105	adc im
6A	106	ror a
6B	107	asr im
6C	108	jmp ()
6D	109	adc ab
6E	110	ror ab
6F	111	rra ab
70	112	bvs rel
71	113	adc iy
72	114	kil

Opcode hex	dez	Befehl
73	115	rra iy
74	116	dop
75	117	adc zx
76	118	ror zx
77	119	rra zx
78	120	sei
79	121	adc ay
7A	122	nop
7B	123	rra ay
7C	124	top
7D	125	adc ax
7E	126	ror ax
7F	127	rra ax
80	128	dop
81	129	sta ix
82	130	–
83	131	aax ix
84	132	sty zp
85	133	sta zp
86	134	stx zp
87	135	aax zp
88	136	dey
89	137	dop
8A	138	txa
8B	139	aax im
8C	140	sty ab
8D	141	sta ab
8E	142	stx ab
8F	143	aax ab
90	144	bcc rel
91	145	sta iy
92	146	kil
93	147	dop
94	148	sty zx
95	149	sta zx
96	150	stx zy
97	151	aax zy
98	152	tya
99	153	sta ay
9A	154	txs

Opcode hex	dez	Befehl
9B	155	–
9C	156	a11 ax
9D	157	sta ax
9E	158	a11 ay
9F	159	–
A0	160	ldy im
A1	161	lda ix
A2	162	ldx im
A3	163	lax ix
A4	164	ldy zp
A5	165	lda zp
A6	166	ldx zp
A7	167	lax zp
A8	168	tay
A9	169	lda im
AA	170	tax
AB	171	–
AC	172	ldy ab
AD	173	lda ab
AE	174	ldx ab
AF	175	ax ab
B0	176	bcs rel
B1	177	lda iy
B2	178	kil
B3	179	lax iy
B4	180	ldy zx
B5	181	lda zx
B6	182	ldx zy
B7	183	lax zy
B8	184	clv
B9	185	lda ay
BA	186	tsx
BB	187	lar ay
BC	188	ldy ax
BD	189	lda ax
BE	190	ldx ay
BF	191	lax ay
C0	192	cpy im
C1	193	cmp ix
C2	194	–

Opcode hex	dez	Befehl
C3	195	dcp ix
C4	196	cpy zp
C5	197	cmp zp
C6	198	dec zp
C7	199	dcp zp
C8	200	iny
C9	201	cmp im
CA	202	dex
CB	203	axs im
CC	204	cpy ab
CD	205	cmp ab
CE	206	dec ab
CF	207	dcp ab
D0	208	bne rel
D1	209	cmp iy
D2	210	kil
D3	211	dcp iy
D4	212	dop
D5	213	cmp zx
D6	214	dec zx
D7	215	dcp zx

Opcode hex	dez	Befehl
D8	216	cld
D9	217	cmp ay
DA	218	nop
DB	219	dcp ay
DC	220	top
DD	221	cmp ax
DE	222	dec ax
DF	223	dcp ax
E0	224	cpx im
E1	225	sbc ix
E2	226	–
E3	227	isc ix
E4	228	cpx zp
E5	229	sbc zp
E6	230	inc zp
E7	231	isc zp
E8	232	inx
E9	233	sbc im
EA	234	nop
EB	235	–
EC	236	cpx ab

Opcode hex	dez	Befehl
ED	237	sbc ab
EE	238	inc ab
EF	239	isc ab
F0	240	beq rel
F1	241	sbc iy
F2	242	kil
F3	243	isc iy
F4	244	dop
F5	245	sbc zx
F6	246	inc zx
F7	247	isc zx
F8	248	sed
F9	249	sbc ay
FA	250	nop
FB	251	isc ay
FC	252	top
FD	253	sbc ax
FE	254	inc ax
FF	255	isc ax

Die gerasterten Opcodes sind illegal.

Adressierungsarten:

= implizit
a = Akkumulator
rel = relativ
zp = Zeropage absolut
zx = Zeropage X-indiziert
tzy = Zeropage Y-indiziert
ab = absolut
ax = absolut X-indiziert
ay = absolut Y-indiziert
( ) = indirekter Sprung
ix = indiziert indirekt
iy = indirekt indizier
im = unmittelbar

Anhang 4

Beeinflussung der Prozessor-Flags

Befehl	N	V	B	D	I	Z	C
all	x					x	
aax	x					x	
adc	x	x				x	
and	x					x	
asl	x					x	x
asr	0					x	x
arr	x					x	x
axs	x	x				x	x
bcc							
bcs							
beq							
bit	7	6				x	
bmi							
bne							
bpl							
brk			1		1		
bvc							
bvs							
clc							0
cld				0			
cli					0		
clv		0					
cmp	x					x	x
cpx	x					x	x
cpy	x					x	x
dcp	x					x	x
dec	x					x	
dex	x					x	

Befehl	**N**	**V**	**B**	**D**	**I**	**Z**	**C**
dey	x					x	
dop							
eor	x					x	
inc	x					x	
inx	x					x	
iny	x					x	
isc	x	x				x	x
jmp							
jsr							
kil							
lar	x					x	
lax	x					x	
lda	x					x	
ldx	x					x	
ldy	x					x	
lsr	0					x	x
nop							
nop							
ora	x					x	
pha							
php							
pla	x					x	
plp	x	x	x	x	x	x	x
rla	x					x	x
rol	x					x	x
ror	x					x	x
rra	x	x				x	x
rti	x	x	x	x	x	x	x
rts							
sbc	x	x				x	x
sec							1
sed				1			
sei					1		
slo	x					x	x
sre	x					x	x
sta							
stx							
sty							
tax	x						x
tay	x						x
tsx	x						x

Befehl	**N**	**V**	**B**	**D**	**I**	**Z**	**C**
txa	x						x
txs							
tya	x						x

Die gerasterten Opcodes sind illegal.

Es bedeuten:

x – Beeinflussung je nach Operation
0 – Flag wird gelöscht
1 – Flag wird gesetzt
6 – Bit 6 des Operanden erscheint im Statusregister
7 – Bit 7 des Operanden erscheint im Statusregister

Anhang 5

Routinen für Kooperation von Basic und Maschinensprache

Adresse	Funktion	Eingabe		Ausgabe	
		Format	Adresse	Format	Adresse
$0073	Holt nächstes Byte	1 Byte	Basic-Text	1 Byte	A
$0079	Holt aktuelles Byte	1 Byte	Basic-Text	1 Byte	A
$A96B	Holt Integerwert (0–63999)	ASCII-Zahl	Basic-Text	2 Byte Integer	$14/ $15
$AD8A	Holt beliebigen numerischen Ausdruck	Basic-Ausdruck	Basic-Text	FLPT	FAC
$AD9E	Holt beliebigen Ausdruck	Basic-Ausdruck	Basic-Text	a) numerisch: FLPT	FAC
				b) String: Zeiger auf Descriptor	FAC+3 / FAC+4

Es werden folgende Flags gesetzt:

$0D: $00 = Zahl, $FF = String
$0E: $00 = Fließkomma, $80 = Integer

$AEF7	Prüft auf »)»	ASCII	Basic-Text	–	–
$AEFA	Prüft auf »(«	ASCII	Basic-Text	–	–
$AEFD	Prüft auf »,«	ASCII	Basic-Text	–	–
$AEFF	Prüft auf Zeichen im Akkumulator	ASCII	Basic-Text	–	–

Die letzten 4 Routinen überlesen das Zeichen, wenn vorhanden.
Sonst wird eine Fehlermeldung »Syntax Error« erzeugt.

Adresse	Funktion	Eingabe		Ausgabe	
		Format	Adresse	Format	Adresse
$AF28	Sucht Variablenwert	Name + Kennung	$45/$46	a) Zahl: FLPT b) String: Zeiger	FAC FAC+3 /FAC+4
$B0E7	Sucht Variablennamen	Name + Kennung	$45/$46	Adresse	$47/ $48
$B79B	Holt Zahl (0–255)	ASCII	Basic-Text	1 Byte	X
$B7EB	Holt 2 Integerzahlen (Trennung durch »,«) 1. Zahl: 0 – 65535 2. Zahl: 0 – 255	ASCII	Basic-Text	2 Byte Integer 1 Byte Integer	$14/ $15 X
$B7F7	Prüft Ausdruck auf 0–65535 nach $AD8A	Basic-Ausdruck	Basic-Text	2 Byte Integer	$14/ $15
$E200	Prüft auf »,« und holt folgende Zahl	ASCII	Basic-Text	1 Byte	X

Anhang 6

Betriebssystemroutinen des C64

			Vorbereitung		
Adresse	Funktion		A	X	Y
$FFB1	LISTEN senden		GA	–	–
$FF93	Sekundäradresse nach LISTEN senden		SA+$60	–	–
$FFAE	UNLISTEN senden		–	–	–
$FFB4	TALK senden		GA	–	–
$FF96	Sekundäradresse nach TALK senden		SA+$60	–	–
$FFAB	UNTALK senden		–	–	–
$FFA5	Zeichen vom seriellen Bus holen in A		–	–	–
$FFA8	Zeichen auf seriellen Bus ausgeben		ASCII	–	–
$FFB7	Status holen in A		–	–	–
$FFBA	Fileparameter setzen		LF	GA	SA
$FFBD	Filenamenparameter setzen		L	L(S)	H(S)
$FFC0	OPEN	$FFBA, $FFBD	–	–	–
$FFC3	CLOSE		LF	–	–
$FFC6	CHKIN Eingabegerät setzen		–	LF	–
$FFC9	CKOUT Ausgabegerät setzen		–	LF	–
$FFCC	CLRCH Ein-/Ausgabegerät rücksetzen		–	–	–
$FFCF	BASIN Eingabe eines Zeichens in A		–	–	–
$FFD2	BSOUT Zeichen ausgeben		ASCII	–	–
$FFD5	LOAD absolut	SA=1,$FFBA,$FFBD	#00	–	–
$FFD5	LOAD nach Adresse	SA=0,$FFBA,$FFBD	#00	L(S)	H(S)
$FFD5	VERIFY	$FFBA,$FFBD	#01	L(S)	H(S)
$FFD8	SAVE	START=L(S),START+1=H(S)	#START	L(E+1)	H(E+1)
$FFE4	GET Eingabe eines Zeichens in A		–	–	–
$FFE7	Schließen aller Ein-/Ausgabekanäle		–	–	–

Es bedeuten:

A	= Akkumulator	X	= X-Register
LF	= logische Filenummer	S	= Startadresse
GA	= Geräteadresse	E	= Endadresse
SA	= Sekundäradresse	L( )	= Lowbyte des Argumentes
ASCII	= ASCII-Code	H( )	= Highbyte des Argumentes
START	= beliebige Zeropage-Adresse	Y	= Y-Register

*: Nach der Ausführung der VERIFY-Routine erkennt man an Hand des Statusbytes, ob der Speicherinhalt mit dem File übereinstimmt: Ist dies der Fall, sind alle Bits außer Bit 6 gleich Null, d.h., der Status nimmt den Wert 64 an.

Anhang 7

Befehlsübersicht Hypra-Ass-Plus

a) Editorbefehle

/A 100,10	Automatische Zeilennumerierung
/O	Re-New des Quelltextes
/D 100-200	Löschen von Zeilenbereichen
/E 100-200	Listen von Zeilenbereichen
/TO,12	Setzen des Tabulators für Assemblerbefehle
/T1,21	Setzen des Tabulators für Kommentare
/T2,3	Anzahl Blanks am Anfang einer Zeile
/T3,15	Setzen des Tabulators für Symboltabelle
/X	Verlassen des Assemblers
/P1,100,200	Setzen eines Arbeitsbereiches (Page)
/ziffer(n)	Formatiertes Listen einer Page
/N1,100,10	Neues Durchnumerieren einer Page
/F1,»string«	Suchen einer Zeichenkette in einer Page
/R2,»str1«,»str2«	Ersetzen von »str2« durch »str1«
/U 14000	Setzen des Quelltextstartes
/B	Anzeige der aktuellen Speicherkonfiguration
/Z 100	Setzen des Quelltextstartes an eine Zeile
/W 100-200,30	Verschieben von Quelltextzeilen
/L »name«	Laden eines Quelltextes
/S (10-30,)»name«	Speichern eines Quelltextes oder eines Teils
/V »name«	Verify eines Quelltextes
/M »name«	Anhängen eines Quelltextes
/G 9	Setzen der Geräteadresse
/I	Lesen und Anzeige des Directorys
/K	Lesen des Fehlerkanals und Anzeige bei Fehler
/	Übermittlung von Diskettenbefehlen
/CH 15	Setzen der Hintergrundfarbe
/CR 6	Setzen der Rahmenfarbe

/!	Anzeige der Symboltabelle (unsortiert)
/!!	Anzeige der Symboltabelle (sortiert)

b) Pseudo-Opcodes

.BA $C000	Definiert Startadresse des Maschinenprogramms
.EQ label=$FFD2	Weist einem Label einen Wert zu
.GL marke=$FFD5	Weist einem globalen Label einen Wert zu
.BY 1,$12,%01,»a«	Einfügen von Ein-Byte-Werten in den Quelltext
.WO $CFDE,label	Einfügen von Adressen im Low/High-Format
.TX »text«	Einfügen von Texten in den Quelltext
.AP »name«	Verketten von Quelltexten
.OB »name,p,w«	Senden des Objektcodes zur Floppy
.EN	Schließen des Objektfiles
.ON ausdruck,jump	Bedingter Sprung, wenn Ausdruck wahr ist
.GO jump	Unbedingter Sprung zu Zeile »jump«
.IF ausdruck	Fortführung der Assemblierung bei .EL, falls Ausdruck falsch ist Sonst bis zu .EL oder .EI
.EL	ELSE-Alternative zu den Zeilen hinter .IF
.EI	Ende der .IF-Konstruktion
.CO var1,var2	Übergabe von Labeln und Quelltext an nachzuladende Teile
.MA makro(p1,p2)	Makrodefinitionszeile
.RT	Ende der Makrodefinition
...makro(p1,p2)	Makroaufruf
.LI lf,ga,sa	Senden von formatierten Listings
.SY lf,ga,sa	Senden der formatierten Symboltabelle
.ST	Beenden der Assemblierung
.DP t0,t1,t2,t3	Setzen der Tabulatoren aus dem Quelltext

c) Befehle des Reassemblers

P $C000	Einsprungpunkt durch Label markieren
T $C000,$C200	Tabelle definieren
E byte	Startet den Reassembler. Die einzelnen Bits des Bytes haben folgende Bedeutung:
Bit 0 gesetzt	Zeropage-Adressen-Label bekommen drei Zeichen
Bit 1 gesetzt	Nach RTS, RTI, BRK, JMP Kommentarzeile einfügen
Bit 2 gesetzt	Bei unmittelbarer Adress. ASCII-Code ausgeben
Bit 3 gesetzt	Kommentarzeilen in Tabellen einfügen
Bit 4 gesetzt	Der ASCII-Code wird bei Tabellen unterdrückt
Bit 5 gesetzt	Externe und Tabellenlabel kennzeichnen
Bit 6 gesetzt	Selbständig nach Tabellen suchen
Bit 7 gesetzt	Auf Speicher unter dem ROM zugreifen

Anhang 8

Befehlsübersicht SMON Plus

a) Befehle aller Versionen

A 4000	Startadresse Zeilenassembler
B 4000 4200	Erzeugung von Basic-Data-Zeilen
C 4010 4200 4013 4000 4200	Verschieben eines Programms mit Adreßumrechnung
D 4000 (4100)	Disassemblierung
F 12,4000 4200	Suche nach einzelnen Hex-Zahlen
FA C000,4000 4050	Suche nach absoluten Adressen
FR 4034,4000 4050	Suche nach relativen Sprüngen
FT 4000 5000	Suche nach Tabellen
FZ A2,4000 4050	Suche nach Zeropage-Adressen
FI 01,4000 4050	Suche nach unmittelbarer Adressierung
GO 4000	Start von Maschinenprogrammen
IO9	Geräteadresse umstellen
K 4000 5000	Gibt Speicherbereich als ASCII-Zeichen aus
L»name« (4000)	Laden an die richtige (angegebene) Adresse
M 4000 (4200)	Ausgabe als Hex-Byte und ASCII-Code
O 4000 4100 23	Füllen eines Bereichs mit einem Byte
PO5	Setzen der Drucker-Gerätenummer
R	Anzeige der Registerinhalte
S»name« 4000 5000	Abspeichern eines Programms
TW (4000)	Einzelschrittmodus. Mit »J« Echtzeitausführung
TB 4100 05	Breakpoint setzen (nach dem 5. Durchlauf)
TQ 4000	Schnellschrittmodus. Springt bei Erreichen eines Breakpoints in die Registeranzeige
TS 4000 4020	Arbeitet ein Programm ab $4000 in Echtzeit ab und springt bei $4020 in die Registeranzeige
V 6000 6200 4000 4100 4200	Ändert alle absoluten Adressen $4100 bis $4200, die sich auf den Bereich von $6000 bis $6200 beziehen, auf den neuen Bereich ab $4000

W 4000 4300 5000	Verschieben von Speicherbereichen
X	Monitor verlassen
#49152	Dezimalzahl umrechnen
$002B	Hex-Zahl umrechnen
%10101011	Binärzahl umrechnen
?0321+3567	Addition zweier Hex-Zahlen
= 4000 5000	Vergleichen von Speicherinhalten

b) Spezielle Befehle des SMON Floppy

Z	Ruft den Diskettenmonitor auf
R (12 01)	Liest Track $12, Sektor $01. Fehlt die Angabe, wird der logisch folgende Block gelesen.
W (12 01)	Schreibt Track $12, Sektor $01. Fehlt die Angabe, werden die Angaben von »R« benutzt.
M	Zeigt den Pufferinhalt als Hex-Bytes
X	Rücksprung zum Monitor

c) Spezielle Befehle des SMON Plus

Z 4000 (5000)	Gibt Bereich binär aus (ein Byte pro Zeile)
H 4000 (5000)	Gibt Bereich binär aus (drei Byte pro Zeile)
N 4000 (5000)	Ausgabe in Bildschirmcode (32 Zeichen pro Zeile)
U 4000 (5000)	Ausgabe in Bildschirmcode (40 Zeichen pro Zeile)
E 4000 (5000)	Füllen des Speicherbereichs mit $00
Y 40	Verschieben des SMON nach $4000
Q 2000	Kopieren des Zeichensatzes nach $2000
J	Bringt den letzten Ausgabebefehl zurück

d) Spezielle Befehle des SMON Illegal

Disassemblierung von folgenden illegalen Opcodes:

- LAX entspricht LDA und LDX
- DCP entspricht DEC und CMP
- ISC entspricht INC und SBC
- RLA entspricht ROL und AND
- RRA entspricht ROR und ADC
- SLO entspricht ASL und ORA
- SRE entspricht LSR und EOR
- SAX entspricht A AND X, STA
- CRA Führt zum Absturz des Prozessors
- NOP Ein-, Zwei- oder Drei-Byte-NOP

Die in Klammern angegebenen Werte können, müssen aber nicht mit eingegeben werden.

Anhang 9

Adressen und Token der Befehle, Funktionen und Operatoren

a) Basic-Befehle

Token	Adresse	Befehl	Token	Adresse	Befehl
$80	$A831	END	$92	$B82D	WAIT
$81	$A742	FOR	$93	$E168	LOAD
$82	$AD1E	NEXT	$94	$E156	SAVE
$83	$A8F8	DATA	$95	$E165	VERIFY
$84	$ABA5	INPUT#	$96	$B3B3	DEF
$85	$ABBF	INPUT	$97	$B824	POKE
$86	$B081	DIM	$98	$AA80	PRINT#
$87	$AC06	READ	$99	$AAA0	PRINT
$88	$A9A5	LET	$9A	$A857	CONT
$89	$A8A0	GOTO	$9B	$A69C	LIST
$8A	$A871	RUN	$9C	$A65E	CLR
$8B	$A928	IF	$9D	$AA86	CMD
$8C	$A81D	RESTORE	$9E	$E12A	SYS
$8D	$A883	GOSUB	$9F	$E1BE	OPEN
$8E	$A8D2	RETURN	$A0	$E1C7	CLOSE
$8F	$A93B	REM	$A1	$AB7B	GET
$90	$A82F	STOP	$A2	$A642	NEW
$91	$A94B	ON			

b) Basic-Funktionen

Token	Adresse	Befehl	Token	Adresse	Befehl
$B4	$BC39	SGN	$B8	$B37D	FRE
$B5	$BCCC	INT	$B9	$B39E	POS
$B6	$BC58	ABS	$BA	$BF71	SQR
$B7	$0310	USR	$BB	$E097	RND

Token	Adresse	Befehl	Token	Adresse	Befehl
$BC	$B9EA	LOG	$C4	$B465	STR$
$BD	$BFED	EXP	$C5	$B7AD	VAL
$BE	$E264	COS	$C6	$B78B	ASC
$BF	$E26B	SIN	$C7	$B6EC	CHR$
$C0	$E2B4	TAN	$C8	$B700	LEFT$
$C1	$E30E	ATN	$C9	$B72C	RIGHT$
$C2	$B80D	PEEK	$CA	$B737	MID$
$C3	$B77C	LEN			

c) Basic-Operatoren

Token	Adresse	Befehl	Token	Adresse	Befehl
$79	$B86A	+	$50	$AFE9	AND
$79	$B853	–	$46	$AFE6	OR
$7B	$BA2B	*	$7D	$BFB4	+ = –
$7B	$BB12	/	$5A	$AED4	Vorz.-Wechsel
$7F	$BF7B	^	$64	$B016	Vergleich

Anhang 10

Die Codes des C64

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN		
@	@	"($0)"	0	00	0000 0000	128	80	1000 0000					
A	a	"($1)"	1	01	0000 0001	129	81	1000 0001			RV-A	CT-A	
B	b	"($2)"	2	02	0000 0010	130	82	1000 0010			RV-B	CT-B	
C	c	"($3)"	3	03	0000 0011	131	83	1000 0011			RV-C	CT-C	STOP
D	d	"($4)"	4	04	0000 0100	132	84	1000 0100			RV-D	CT-D	
E	e	"(WHT)"	5	05	0000 0101	133	85	1000 0101			RV-E	CT-E	CT-2
F	f	"($6)"	6	06	0000 0110	134	86	1000 0110			RV-F	CT-F	CT-←
G	g	"($7)"	7	07	0000 0111	135	87	1000 0111			RV-G	CT-G	
H	h	"(DISH)"	8	08	0000 1000	136	88	1000 1000			RV-H	CT-H	
I	i	"(ENSH)"	9	09	0000 1001	137	89	1000 1001			RV-I	CT-I	
J	j	"($10)"	10	0A	0000 1010	138	8A	1000 1010			RV-J	CT-J	
K	k	"($11)"	11	0B	0000 1011	139	8B	1000 1011			RV-K	CT-K	
L	l	"($12)"	12	0C	0000 1100	140	8C	1000 1100			RV-L	CT-L	
M	m	"(CR)"	13	0D	0000 1101	141	8D	1000 1101			RV-M	CT-M	CR*
N	n	"(SWLC)"	14	0E	0000 1110	142	8E	1000 1110			RV-N	CT-N	
O	o	"($15)"	15	0F	0000 1111	143	8F	1000 1111			RV-O	CT-O	
P	p	"($16)"	16	10	0001 0000	144	90	1001 0000			RV-P	CT-P	
Q	q	"(DOWN)"	17	11	0001 0001	145	91	1001 0001			RV-Q	CT-Q	DOWN
R	r	"(RVS)"	18	12	0001 0010	146	92	1001 0010			RV-R	CT-R	CT-9
S	s	"(HOME)"	19	13	0001 0011	147	93	1001 0011			RV-S	CT-S	HOME
T	t	"(DEL)"	20	14	0001 0100	148	94	1001 0100			RV-T	CT-T	DEL
U	u	"($21)"	21	15	0001 0101	149	95	1001 0101			RV-U	CT-U	
V	v	"($22)"	22	16	0001 0110	150	96	1001 0110			RV-V	CT-V	
W	w	"($23)"	23	17	0001 0111	151	97	1001 0111			RV-W	CT-W	
X	x	"($24)"	24	18	0001 1000	152	98	1001 1000			RV-X	CT-X	
Y	y	"($25)"	25	19	0001 1001	153	99	1001 1001			RV-Y	CT-Y	
Z	z	"($26)"	26	1A	0001 1010	154	9A	1001 1010			RV-Z	CT-Z	
[	[	"(ESC)"	27	1B	0001 1011	155	9B	1001 1011			RV-SH-:	CT-:	RV-C=-:
£	£	"(RED)"	28	1C	0001 1100	156	9C	1001 1100			RV-£	CT-£	CT-3
]	]	"(RGHT)"	29	1D	0001 1101	157	9D	1001 1101			RV-SH-;	CT-;	RGHT
↑	↑	"(GRN)"	30	1E	0001 1110	158	9E	1001 1110			RV-↑	CT-↑	CT-6
←	←	"(BLU)"	31	1F	0001 1111	159	9F	1001 1111			RV-←	CT-=	CT-7
□	□	□	32	20	0010 0000	32	20	0010 0000	160	160	SP		
■	■					160	A0	1010 0000			RV-SP		
!	!	!	33	21	0010 0001	33	21	0010 0001			SH-1		
!	!					161	A1	1010 0001			RV-SH-1		

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM ASCII	TASTENKOMBINATIONEN		
"	"	"	34	22	0010 0010	34	22	0010 0010		SH-2		
"	"					162	A2	1010 0010		RV-SH-2		
#	#	#	35	23	0010 0011	35	23	0010 0011		SH-3		
#	#					163	A3	1010 0011		RV-SH-3		
$	$	$	36	24	0010 0100	36	24	0010 0100		SH-4		
$	$					164	A4	1010 0100		RV-SH-4		
%	%	%	37	25	0010 0101	37	25	0010 0101		SH-5		
%	%					165	A5	1010 0101		RV-SH-5		
&	&	&	38	26	0010 0110	38	26	0010 0110		SH-6		
&	&					166	A6	1010 0110		RV-SH-6		
'	'	'	39	27	0010 0111	39	27	0010 0111		SH-7		
'	'					167	A7	1010 0111		RV-SH-7		
(	(	(	40	28	0010 1000	40	28	0010 1000		SH-8		
(	(					168	A8	1010 1000		RV-SH-8		
)	)	)	41	29	0010 1001	41	29	0010 1001		SH-9	C=-9	
)	)					169	A9	1010 1001		RV-SH-9		
*	*	*	42	2A	0010 1010	42	2A	0010 1010		*		
*	*					170	AA	1010 1010		RV-*		
+	+	+	43	2B	0010 1011	43	2B	0010 1011		+		
+	+					171	AB	1010 1011		RV-+		
,	,	,	44	2C	0010 1100	44	2C	0010 1100		,		
,	,					172	AC	1010 1100		RV-,		
-	-	-	45	2D	0010 1101	45	2D	0010 1101		-		
-	-					173	AD	1010 1101		RV--		
.	.	.	46	2E	0010 1110	46	2E	0010 1110		.		
.	.					174	AE	1010 1110		RV-.		
/	/	/	47	2F	0010 1111	47	2F	0010 1111		/		
/	/					175	AF	1010 1111		RV-/		
0	0	0	48	30	0011 0000	48	30	0011 0000		0	SH-0	C=-0
0	0					176	B0	1011 0000		RV-0	RV-SH-0	RV-C=-0
1	1	1	49	31	0011 0001	49	31	0011 0001		1		
1	1					177	B1	1011 0001		RV-1		
2	2	2	50	32	0011 0010	50	32	0011 0010		2		
2	2					178	B2	1011 0010		RV-2		
3	3	3	51	33	0011 0011	51	33	0011 0011		3		
3	3					179	B3	1011 0011		RV-3		
4	4	4	52	34	0011 0100	52	34	0011 0100		4		
4	4					180	B4	1011 0100		RV-4		
5	5	5	53	35	0011 0101	53	35	0011 0101		5		
5	5					181	B5	1011 0101		RV-5		
6	6	6	54	36	0011 0110	54	36	0011 0110		6		
6	6					182	B6	1011 0110		RV-6		
7	7	7	55	37	0011 0111	55	37	0011 0111		7		
7	7					183	B7	1011 0111		RV-7		
8	8	8	56	38	0011 1000	56	38	0011 1000		8		
8	8					184	B8	1011 1000		RV-8		
9	9	9	57	39	0011 1001	57	39	0011 1001		9		
9	9					185	B9	1011 1001		RV-9		
:	:	:	58	3A	0011 1010	58	3A	0011 1010		:		
:	:					186	BA	1011 1010		RV-:		
;	;	;	59	3B	0011 1011	59	3B	0011 1011		;		
;	;					187	BB	1011 1011		RV-;		
<	<	<	60	3C	0011 1100	60	3C	0011 1100		SH-,	C=-,	
<	<					188	BC	1011 1100		RV-SH-,	RV-C=-,	

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN		
=	=	=	61	3D	0011 1101	61	3D	0011 1101			=	SH-=	C=-=
= (revers)	= (revers)					189	BD	1011 1101			RV-=	RV-SH-=	RV-C=-=
>	>	>	62	3E	0011 1110	62	3E	0011 1110			SH-.	C=-.	
> (revers)	> (revers)					190	BE	1011 1110			RV-SH-.	RV-C=-.	
?	?	?	63	3F	0011 1111	63	3F	0011 1111			SH-/	C=-/	
? (revers)	? (revers)					191	BF	1011 1111			RV-SH-/	RV-C=-/	
@	@	@	64	40	0100 0000	0	00	0000 0000			@		
A	a	A	65	41	0100 0001	1	01	0000 0001		193	A		
B	b	B	66	42	0100 0010	2	02	0000 0010		194	B		
C	c	C	67	43	0100 0011	3	03	0000 0011		195	C		
D	d	D	68	44	0100 0100	4	04	0000 0100		196	D		
E	e	E	69	45	0100 0101	5	05	0000 0101		197	E		
F	f	F	70	46	0100 0110	6	06	0000 0110		198	F		
G	g	G	71	47	0100 0111	7	07	0000 0111		199	G		
H	h	H	72	48	0100 1000	8	08	0000 1000		200	H		
I	i	I	73	49	0100 1001	9	09	0000 1001		201	I		
J	j	J	74	4A	0100 1010	10	0A	0000 1010		202	J		
K	k	K	75	4B	0100 1011	11	0B	0000 1011		203	K		
L	l	L	76	4C	0100 1100	12	0C	0000 1100		204	L		
M	m	M	77	4D	0100 1101	13	0D	0000 1101		205	M		
N	n	N	78	4E	0100 1110	14	0E	0000 1110		206	N		
O	o	O	79	4F	0100 1111	15	0F	0000 1111		207	O		
P	p	P	80	50	0101 0000	16	10	0001 0000		208	P		
Q	q	Q	81	51	0101 0001	17	11	0001 0001		209	Q		
R	r	R	82	52	0101 0010	18	12	0001 0010		210	R		
S	s	S	83	53	0101 0011	19	13	0001 0011		211	S		
T	t	T	84	54	0101 0100	20	14	0001 0100		212	T		
U	u	U	85	55	0101 0101	21	15	0001 0101		213	U		
V	v	V	86	56	0101 0110	22	16	0001 0110		214	V		
W	w	W	87	57	0101 0111	23	17	0001 0111		215	W		
X	x	X	88	58	0101 1000	24	18	0001 1000		216	X		
Y	y	Y	89	59	0101 1001	25	19	0001 1001		217	Y		
Z	z	Z	90	5A	0101 1010	26	1A	0001 1010		218	Z		
[	[	[	91	5B	0101 1011	27	1B	0001 1011			SH-:	C=-:	
£	£	\	92	5C	0101 1100	28	1C	0001 1100			£		
]	]	]	93	5D	0101 1101	29	1D	0001 1101			SH-;	C=-;	
↑	↑	^	94	5E	0101 1110	30	1E	0001 1110			↑		
←	←	_	95	5F	0101 1111	31	1F	0001 1111			←	SH-←	C=-←
─	─	`	96	60	0110 0000	64	40	0100 0000	192				
♠	A	a	97	61	0110 0001	65	41	0100 0001	193				
│	B	b	98	62	0110 0010	66	42	0100 0010	194				
─	C	c	99	63	0110 0011	67	43	0100 0011	195				
▔	D	d	100	64	0110 0100	68	44	0100 0100	196				
▔	E	e	101	65	0110 0101	69	45	0100 0101	197				
─	F	f	102	66	0110 0110	70	46	0100 0110	198				
│	G	g	103	67	0110 0111	71	47	0100 0111	199				
│	H	h	104	68	0110 1000	72	48	0100 1000	200				
╮	I	i	105	69	0110 1001	73	49	0100 1001	201				
╰	J	j	106	6A	0110 1010	74	4A	0100 1010	202				
╯	K	k	107	6B	0110 1011	75	4B	0100 1011	203				
└	L	l	108	6C	0110 1100	76	4C	0100 1100	204				
╲	M	m	109	6D	0110 1101	77	4D	0100 1101	205				
╱	N	n	110	6E	0110 1110	78	4E	0100 1110	206				
┌	O	o	111	6F	0110 1111	79	4F	0100 1111	207				

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN		
	P	p	112	70	0111 0000	80	50	0101 0000	208				
	Q	q	113	71	0111 0001	81	51	0101 0001	209				
	R	r	114	72	0111 0010	82	52	0101 0010	210				
	S	s	115	73	0111 0011	83	53	0101 0011	211				
	T	t	116	74	0111 0100	84	54	0101 0100	212				
	U	u	117	75	0111 0101	85	55	0101 0101	213				
	V	v	118	76	0111 0110	86	56	0101 0110	214				
	W	w	119	77	0111 0111	87	57	0101 0111	215				
	X	x	120	78	0111 1000	88	58	0101 1000	216				
	Y	y	121	79	0111 1001	89	59	0101 1001	217				
	Z	z	122	7A	0111 1010	90	5A	0101 1010	218				
			123	7B	0111 1011	91	5B	0101 1011	219				
			124	7C	0111 1100	92	5C	0101 1100	220				
			125	7D	0111 1101	93	5D	0101 1101	221				
			126	7E	0111 1110	94	5E	0101 1110	222				
		"($127)"	127	7F	0111 1111	95	5F	0101 1111	223				
		"($128)"	128	80	1000 0000	192	C0	1100 0000			RV-SH-£		
	A	"(ORNG)"	129	81	1000 0001	193	C1	1100 0001			RV-SH-A	C=-1	
	B	"($130)"	130	82	1000 0010	194	C2	1100 0010			RV-SH-B		
	C	"($131)"	131	83	1000 0011	195	C3	1100 0011			RV-SH-C	SH-STOP*	C=-STOP*
	D	"($132)"	132	84	1000 0100	196	C4	1100 0100			RV-SH-D		
	E	"(F1)"	133	85	1000 0101	197	C5	1100 0101			RV-SH-E	F1	
	F	"(F3)"	134	86	1000 0110	198	C6	1100 0110			RV-SH-F	F3	
	G	"(F5)"	135	87	1000 0111	199	C7	1100 0111			RV-SH-G	F5	
	H	"(F7)"	136	88	1000 1000	200	C8	1100 1000			RV-SH-H	F7	
	I	"(F2)"	137	89	1000 1001	201	C9	1100 1001			RV-SH-I	SH-F1	C=-F1
	J	"(F4)"	138	8A	1000 1010	202	CA	1100 1010			RV-SH-J	SH-F3	C=-F3
	K	"(F6)"	139	8B	1000 1011	203	CB	1100 1011			RV-SH-K	SH-F5	C=-F5
	L	"(F8)"	140	8C	1000 1100	204	CC	1100 1100			RV-SH-L	SH-F7	C=-F7
	M	"(SHRT)"	141	8D	1000 1101	205	CD	1100 1101			RV-SH-M	SH-CR*	C=-CR*
	N	"(SWUC)"	142	8E	1000 1110	206	CE	1100 1110			RV-SH-N		
	O	"($143)"	143	8F	1000 1111	207	CF	1100 1111			RV-SH-O		
	P	"(BLK)"	144	90	1001 0000	208	D0	1101 0000			RV-SH-P	CT-1	
	Q	"(UP)"	145	91	1001 0001	209	D1	1101 0001			RV-SH-Q	SH-DOWN	C=-DOWN
	R	"(OFF)"	146	92	1001 0010	210	D2	1101 0010			RV-SH-R	CT-0	
	S	"(CLR)"	147	93	1001 0011	211	D3	1101 0011			RV-SH-S	SH-HOME	C=-HOME
	T	"(INST)"	148	94	1001 0100	212	D4	1101 0100			RV-SH-T	SH-DEL	C=-DEL
	U	"(BRN)"	149	95	1001 0101	213	D5	1101 0101			RV-SH-U	C=-2	
	V	"(LRED)"	150	96	1001 0110	214	D6	1101 0110			RV-SH-V	C=-3	
	W	"(GRY1)"	151	97	1001 0111	215	D7	1101 0111			RV-SH-W	C=-4	
	X	"(GRY2)"	152	98	1001 1000	216	D8	1101 1000			RV-SH-X	C=-5	
	Y	"(LGRN)"	153	99	1001 1001	217	D9	1101 1001			RV-SH-Y	C=-6	
	Z	"(LBLU)"	154	9A	1001 1010	218	DA	1101 1010			RV-SH-Z	C=-7	
		"(GRY3)"	155	9B	1001 1011	219	DB	1101 1011			RV-SH-+	C=-8	
		"(PUR)"	156	9C	1001 1100	220	DC	1101 1100			RV-C=--	CT-5	
		"(LEFT)"	157	9D	1001 1101	221	DD	1101 1101			RV-SH--	SH-RGHT	C=-RGHT
		"(YEL)"	158	9E	1001 1110	222	DE	1101 1110			RV-SH-↑	CT-8	RV-C=-↑
		"(CYN)"	159	9F	1001 1111	223	DF	1101 1111			RV-C=-£	CT-4	
			160	A0	1010 0000	96	60	0110 0000	224	224	SH-SP	C=-SP	
						224	E0	1110 0000			RV-SH-SP	RV-C=-SP	
			161	A1	1010 0001	97	61	0110 0001	225	225	C=-K		
						225	E1	1110 0001			RV-C=-K		
			162	A2	1010 0010	98	62	0110 0010	226	226	C=-I		
						226	E2	1110 0010			RV-C=-I		
			163	A3	1010 0011	99	63	0110 0011	227	227	C=-T		
						227	E3	1110 0011			RV-C=-T		

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN
			164	A4	1010 0100	100	64	0110 0100	228	228	C=-@
						228	E4	1110 0100			RV-C=-@
			165	A5	1010 0101	101	65	0110 0101	229	229	C=-G
						229	E5	1110 0101			RV-C=-G
			166	A6	1010 0110	102	66	0110 0110	230	230	C=-+
						230	E6	1110 0110			RV-C=-+
			167	A7	1010 0111	103	67	0110 0111	231	231	C=-M
						231	E7	1110 0111			RV-C=-M
			168	A8	1010 1000	104	68	0110 1000	232	232	C=-£
						232	E8	1110 1000			RV-C=-£
			169	A9	1010 1001	105	69	0110 1001	233	233	SH-£
						233	E9	1110 1001			RV-SH-£
			170	AA	1010 1010	106	6A	0110 1010	234	234	C=-N
						234	EA	1110 1010			RV-C=-N
			171	AB	1010 1011	107	6B	0110 1011	235	235	C=-Q
						235	EB	1110 1011			RV-C=-Q
			172	AC	1010 1100	108	6C	0110 1100	236	236	C=-D
						236	EC	1110 1100			RV-C=-D
			173	AD	1010 1101	109	6D	0110 1101	237	237	C=-Z
						237	ED	1110 1101			RV-C=-Z
			174	AE	1010 1110	110	6E	0110 1110	238	238	C=-S
						238	EE	1110 1110			RV-C=-S
			175	AF	1010 1111	111	6F	0110 1111	239	239	C=-P
						239	EF	1110 1111			RV-C=-P
			176	B0	1011 0000	112	70	0111 0000	240	240	C=-A
						240	F0	1111 0000			RV-C=-A
			177	B1	1011 0001	113	71	0111 0001	241	241	C=-E
						241	F1	1111 0001			RV-C=-E
			178	B2	1011 0010	114	72	0111 0010	242	242	C=-R
						242	F2	1111 0010			RV-C=-R
			179	B3	1011 0011	115	73	0111 0011	243	243	C=-W
						243	F3	1111 0011			RV-C=-W
			180	B4	1011 0100	116	74	0111 0100	244	244	C=-H
						244	F4	1111 0100			RV-C=-H
			181	B5	1011 0101	117	75	0111 0101	245	245	C=-J
						245	F5	1111 0101			RV-C=-J
			182	B6	1011 0110	118	76	0111 0110	246	246	C=-L
						246	F6	1111 0110			RV-C=-L
			183	B7	1011 0111	119	77	0111 0111	247	247	C=-Y
						247	F7	1111 0111			RV-C=-Y
			184	B8	1011 1000	120	78	0111 1000	248	248	C=-U
						248	F8	1111 1000			RV-C=-U
			185	B9	1011 1001	121	79	0111 1001	249	249	C=-O
						249	F9	1111 1001			RV-C=-O
			186	BA	1011 1010	122	7A	0111 1010	250	250	SH-@
						250	FA	1111 1010			RV-SH-@
			187	BB	1011 1011	123	7B	0111 1011	251	251	C=-F
						251	FB	1111 1011			RV-C=-F
			188	BC	1011 1100	124	7C	0111 1100	252	252	C=-C
						252	FC	1111 1100			RV-C=-C
			189	BD	1011 1101	125	7D	0111 1101	253	253	C=-X
						253	FD	1111 1101			RV-C=-X
			190	BE	1011 1110	126	7E	0111 1110	254	254	C=-V
						254	FE	1111 1110			RV-C=-V

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN	
			191	BF	1011 1111	127	7F	0111 1111			C=-B	
						255	FF	1111 1111			RV-C=-B	
			192	C0	1100 0000	64	40	0100 0000	96		SH-*	
			193	C1	1100 0001	65	41	0100 0001	97	65	SH-A	
			194	C2	1100 0010	66	42	0100 0010	98	66	SH-B	
			195	C3	1100 0011	67	43	0100 0011	99	67	SH-C	
			196	C4	1100 0100	68	44	0100 0100	100	68	SH-D	
			197	C5	1100 0101	69	45	0100 0101	101	69	SH-E	
			198	C6	1100 0110	70	46	0100 0110	102	70	SH-F	
			199	C7	1100 0111	71	47	0100 0111	103	71	SH-G	
			200	C8	1100 1000	72	48	0100 1000	104	72	SH-H	
			201	C9	1100 1001	73	49	0100 1001	105	73	SH-I	
			202	CA	1100 1010	74	4A	0100 1010	106	74	SH-J	
			203	CB	1100 1011	75	4B	0100 1011	107	75	SH-K	
			204	CC	1100 1100	76	4C	0100 1100	108	76	SH-L	
			205	CD	1100 1101	77	4D	0100 1101	109	77	SH-M	
			206	CE	1100 1110	78	4E	0100 1110	110	78	SH-N	
			207	CF	1100 1111	79	4F	0100 1111	111	79	SH-O	
			208	D0	1101 0000	80	50	0101 0000	112	80	SH-P	
			209	D1	1101 0001	81	51	0101 0001	113	81	SH-Q	
			210	D2	1101 0010	82	52	0101 0010	114	82	SH-R	
			211	D3	1101 0011	83	53	0101 0011	115	83	SH-S	
			212	D4	1101 0100	84	54	0101 0100	116	84	SH-T	
			213	D5	1101 0101	85	55	0101 0101	117	85	SH-U	
			214	D6	1101 0110	86	56	0101 0110	118	86	SH-V	
			215	D7	1101 0111	87	57	0101 0111	119	87	SH-W	
			216	D8	1101 1000	88	58	0101 1000	120	88	SH-X	
			217	D9	1101 1001	89	59	0101 1001	121	89	SH-Y	
			218	DA	1101 1010	90	5A	0101 1010	122	90	SH-Z	
			219	DB	1101 1011	91	5B	0101 1011	123		SH-+	
			220	DC	1101 1100	92	5C	0101 1100	124		C=--	
			221	DD	1101 1101	93	5D	0101 1101	125		SH--	
			222	DE	1101 1110	94	5E	0101 1110	126		SH-↑*	C=-↑*
			223	DF	1101 1111	95	5F	0101 1111	127		C=-*	
			224	E0	1110 0000	96	60	0110 0000	160	160		
						224	E0	1110 0000				
			225	E1	1110 0001	97	61	0110 0001	161	161		
						225	E1	1110 0001				
			226	E2	1110 0010	98	62	0110 0010	162	162		
						226	E2	1110 0010				
			227	E3	1110 0011	99	63	0110 0011	163	163		
						227	E3	1110 0011				
			228	E4	1110 0100	100	64	0110 0100	164	164		
						228	E4	1110 0100				
			229	E5	1110 0101	101	65	0110 0101	165	165		
						229	E5	1110 0101				
			230	E6	1110 0110	102	66	0110 0110	166	166		
						230	E6	1110 0110				
			231	E7	1110 0111	103	67	0110 0111	167	167		
						231	E7	1110 0111				
			232	E8	1110 1000	104	68	0110 1000	168	168		
						232	E8	1110 1000				
			233	E9	1110 1001	105	69	0110 1001	169	169		
						233	E9	1110 1001				
			234	EA	1110 1010	106	6A	0110 1010	170	170		
						234	EA	1110 1010				

ZEICHEN GRAFIK	TEXT	ASCII	ASCII- CHR$- CODE DEZ	HEX	BIN	BILDSCHIRM- CODE DEZ	HEX	BIN	REFERENZ COM	ASCII	TASTENKOMBINATIONEN	
			235	EB	1110 1011	107	6B	0110 1011	171	171		
						235	EB	1110 1011				
			236	EC	1110 1100	108	6C	0110 1100	172	172		
						236	EC	1110 1100				
			237	ED	1110 1101	109	6D	0110 1101	173	173		
						237	ED	1110 1101				
			238	EE	1110 1110	110	6E	0110 1110	174	174		
						238	EE	1110 1110				
			239	EF	1110 1111	111	6F	0110 1111	175	175		
						239	EF	1110 1111				
			240	F0	1111 0000	112	70	0111 0000	176	176		
						240	F0	1111 0000				
			241	F1	1111 0001	113	71	0111 0001	177	177		
						241	F1	1111 0001				
			242	F2	1111 0010	114	72	0111 0010	178	178		
						242	F2	1111 0010				
			243	F3	1111 0011	115	73	0111 0011	179	179		
						243	F3	1111 0011				
			244	F4	1111 0100	116	74	0111 0100	180	180		
						244	F4	1111 0100				
			245	F5	1111 0101	117	75	0111 0101	181	181		
						245	F5	1111 0101				
			246	F6	1111 0110	118	76	0111 0110	182	182		
						246	F6	1111 0110				
			247	F7	1111 0111	119	77	0111 0111	183	183		
						247	F7	1111 0111				
			248	F8	1111 1000	120	78	0111 1000	184	184		
						248	F8	1111 1000				
			249	F9	1111 1001	121	79	0111 1001	185	185		
						249	F9	1111 1001				
			250	FA	1111 1010	122	7A	0111 1010	186	186		
						250	FA	1111 1010				
			251	FB	1111 1011	123	7B	0111 1011	187	187		
						251	FB	1111 1011				
			252	FC	1111 1100	124	7C	0111 1100	188	188		
						252	FC	1111 1100				
			253	FD	1111 1101	125	7D	0111 1101	189	189		
						253	FD	1111 1101				
			254	FE	1111 1110	126	7E	0111 1110	190	190		
						254	FE	1111 1110				
			255	FF	1111 1111	94	5E	0101 1110	126	222	SH-↑	C=-↑

* = Das Zeichen ist mit dieser Tastenkombination nur über die Funktion GET erreichbar.

Fettgedruckte Tastenkombinationen sind nur im Quote Modus möglich

Die Abkürzungen der ASCII-Zeichen bedeuten:

NUL = Null
STX = Start of Text (Textbeginn)
ENQ = Enquiry (Testanfrage)
BEL = Bell (Klingelzeichen)
HT = Horizontal Tabulation
VT = Vertical Tabulation
CR = Carriage Return
SI = Shift in (Shift ein)
SYN = Synchronous (synchronlos)
EOT = End of Transmission (Übertragungsende)
DLE = Data Link Escape (Datenverbindung abhalten)
DC = Device Control (Gerätesteuerung)
NAK = Negative Acknowledge (Gegenquittierung)
ETB = End of Transmission Block (Block-Übertragungsende)
CAN = Cancel (Stornieren, z.B. Puffer leeren)
EM = End of Medium (z.B. Papierende)
ESC = Escape (übergehen, Druckercode)
FS = File Separator (Filetrennung)
GS = Group Separator (Trennung von Gruppen)
RS = Record Separator (Aufzeichnung trennen)
US = Unit Separator (Einheitentrenner)
DEL = Delete (Löschen)
SOH = Start of Heading
ETX = End of Text (Textende)
ACK = Acknowledge (Quittierung)
BS = Backspace (Zeichen zurück)
LF = Line Feed (Zeilenvorschub)
FF = Form Feed (Formatanpassung)
SO = Shift out (Shift aus)
SP = Space (Leerzeichen)
SUB = Substitute (Austausch)

Der Tastencode kann wahlweise aus den Speicherzellen 197 ($C5) oder 203 ($CB) ausgelesen werden.

Stichwortverzeichnis

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X

Y

Z



Haben Sie schon mal Profi-Software zum Buchpreis gekauft?

»Bookware« – das sind professionelle Programme zum Preis eines Buches!



M. Pahl, T. Rullkötter, M. Kuk
C64/C128 MasterText Plus
1988, 201 Seiten, inkl. Diskette
MasterText Plus – die leistungsfähige Textverarbeitung: 40-Zeichen- und 80-Zeichen-Ausgabe – Suchen und Ersetzen – Silbentrennung – Blockoperationen – Formularverwaltung – integrierte Centronics-Schnittstelle – jetzt mit Rechtschreibkorrektur und Adreßverwaltung – Komprimieren von Texten – individuelle Farbgebung und Druckeranpassung – freie Tastenbelegung – Zeichensatz-Editor – komfortable Druckeranpassung: Druckertreiber für MPS 801, MPS 802, Epson-Drucker und Kompatible.
Bestell-Nr. 90527, ISBN 3-89090-527-7
DM 59,–* (sFr 54,30*/öS 502,–*)

S. Baloui
C64/C128 MasterBase
1988, 155 Seiten, inkl. Diskette
Die professionelle Dateiverwaltung für den C64/C128. Besondere Leistungsmerkmale: integrierte Centronics-Schnittstelle – Export und Import von Daten – nachträgliche Veränderung der Struktur einer bereits bestehenden Datei – Tastatur-Makros – einfache Bedienung über Windows und Pull-down-Menüs – als einzige Dateiverwaltung für den C64 erlaubt Ihnen MasterBase, beliebig viele Indexfelder zu verwenden (extrem schnelle Suche nach bestimmten Daten; selbst größte Dateien werden in Nullzeit umsortiert).
Bestell-Nr. 90583, ISBN 3-89090-583-8
DM 59,–* (sFr 54,30*/öS 502,–*)

W. Oppacher, K. Oppacher, M. Wenzel
C64/C128 Giga Paint
1988, ca. 200 Seiten, inkl. 2 Disketten
Ein professionelles Mal- und Zeichenprogramm: stufenloses Verkleinern, Vergrößern und Verzerren – Zeichnen von Kurven durch beliebige Punkte und 3-D-Operationen unter Verwendung aller 16 Farben – Kompatibilität zu über 30 Grafikprogrammen – universelle Druckroutine für fast jeden grafikfähigen Drucker – Ausdruck beliebiger Bildausschnitte – frei definierbare Graustufen – Basic-Erweiterung – beliebige Positionierung von Bildschirmausschnitten – Programmierung flimmerfreier Rasterinterrupts und vieles mehr.
Bestell-Nr. 90619, ISBN 3-89090-619-2
DM 59,–* (sFr 54,30*/öS 502,–*)

* Unverbindliche Preisempfehlung

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0.
SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656.
ÖSTERREICH: Markt&Technik Verlag Gesellschaft m.b.H., Große Neugasse 28, A-1040 Wien, Telefon (0222) 5871393-0;
Rudolf Lechner & Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526

Fragen Sie bei Ihrem Buchhändler nach unserem kostenlosen Gesamtverzeichnis mit über 500 aktuellen Computerbüchern und Software. Oder fordern Sie es direkt beim Verlag an!

Bitte schneiden Sie diesen Coupon aus, und schicken Sie ihn in einem Kuvert an:
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar

Computerliteratur und Software vom Spezialisten

Vom Einsteigerbuch für den Heim- oder Personalcomputer-Neuling über professionelle Programmierhandbücher bis hin zum Elektronikbuch bieten wir Ihnen interessante und topaktuelle Titel für

• Apple-Computer • Atari-Computer • Commodore 64/128/16/116/Plus 4 • Schneider-Computer • IBM-PC, XT und Kompatible
sowie zu den Fachbereichen Programmiersprachen • Betriebssysteme (CP/M, MS-DOS, Unix, Z80) • Textverarbeitung • Datenbanksysteme • Tabellenkalkulation • Integrierte Software • Mikroprozessoren • Schulungen.
Außerdem finden Sie professionelle Spitzen-Programme in unserem preiswerten Software-Angebot für Amiga, Atari ST, Commodore 128, 128 D, 64, 16, für Schneider-Computer und für IBM-PCs und Kompatible!
Fordern Sie mit dem nebenstehenden Coupon unser neuestes Gesamtverzeichnis und unsere Programmservice-Übersichten an, mit hilfreichen Utilities, professionellen Anwendungen oder packenden Computerspielen!



709005

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Adresse:

Name ____

Straße ____

Ort ____

Bitte schicken Sie mir:
☐ Ihr neuestes Gesamtverzeichnis
☐ Eine Übersicht Ihres Programmservice-Angebotes aus der Zeitschrift ____
☐ Außerdem interessiere ich mich für folgende/n Computer: ____

(PS: Wir speichern Ihre Daten und verpflichten uns zur Einhaltung des Bundesdatenschutzgesetzes)

Markt&Technik Verlag AG
– Unternehmensbereich Buchverlag –
Hans-Pinsel-Straße 2
D-8013 Haar bei München

Frank Riemenschneider

C64/C128 Alles über Maschinensprache

Bei kaum einem anderen Heimcomputer ist die Diskrepanz zwischen der Leistungsfähigkeit der Hardware und deren Ausnutzung durch das eingebaute Basic so groß wie beim C64. So nutzen Maschinensprache-Programme gegenüber Basic den gesamten Speicher aus, erreichen eine bis zu 100mal schnellere Verarbeitungsgeschwindigkeit und lassen neue Programmiertechniken wie die Interrupt-Programmierung zu.

Da jetzt aber viele C64-Besitzer sagen: Assembler bzw. Maschinensprache ist mir zu schwierig, zu kompliziert, ging der Autor dieses Buches einen neuen Weg. Er führt den Leser über Theorie und anschließende, vertiefende Praxis zum Erfolg, wobei die nötige Software auf der beiliegenden Diskette enthalten ist.

Im ersten Kapitel finden Sie jeden einzelnen 6510-Prozessorbefehl in Funktion, Wirkung und anhand von Beispielen erklärt. Alle weiteren Kapitel beschäftigen sich, aufgeteilt nach Anwendungsgebieten, mit ausführlichen, praktischen Übungen.

Die Grundlage bildet das beiliegende Assembler-Entwicklungspaket mit den Programmen Hypra-Ass-Plus Makroassembler, SMON-Plus Maschinensprachemonitor, Reassembler und Einzelschrittsimulator.

Folgende Themen werden umfassend beschrieben:

- der 6510-Mikroprozessor
- die Interrupt-Programmierung
- der Variableneinsatz in Maschinensprache
- die HiRes-Grafik-Programmierung
- die Programmierung einer Basic-Erweiterung

Mit den gewonnenen Kenntnissen lassen sich z.B. Hunderte von Datensätzen in Sekunden sortieren, anspruchsvolle Grafiken programmieren, oder Sie richten Ihren C64 mit eigenen Basic-Befehlen auf Ihre individuellen Bedürfnisse ein. Zu allen beschriebenen Anwendungen finden Sie Beispielprogramme, die auch auf der beiliegenden Diskette, im Format 1541, enthalten sind.

Ein umfangreicher Anhang mit Tabellen, Übersichten und Aufstellungen aller wichtigen Daten für die Maschinenprogrammierung wie Befehle, Opcodes, Token, ROM-Routinen rundet das Buch ab.

Hardware-Voraussetzungen:

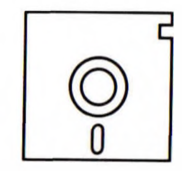
C64 bzw. C128 im 64er-Modus mit einer Floppy (1541, 1570, 1571)

Inhalt der Begleitdiskette:

Hypra-Ass-Plus Makroassembler, SMON-Plus Maschinensprachemonitor, Reassembler, Einzelschrittsimulator, Beispielprogramme.

ISB N 3-89090-571-4







DM 59,-
sFr 54,30
öS 460,20